SPECIFICATION AND ANALYSIS OF TIMING PROPERTIES IN REAL-TIME SYSTEMS

APPROVED BY
SUPERVISORY COMMITTEE:

Aloysius K. Mok

Sanjoy Baruah

Simon S. Lam

Louis Rose

Mitchell Gouda

SPECIFICATION AND ANALYSIS OF TIMING PROPERTIES IN REAL-TIME SYSTEMS

by

FARNAM JAHANIAN, B.S., M.S.C.S.

DISSERTATION

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN
December 1988

To my parents,

Effat and Gholamali Jahanian

Acknowledgements

I would like to thank my advisor, Professor Al Mok, whose support and collaboration made this work possible. He has been a friend and mentor in the full sense of the word. I would also like to express my appreciation to Professor Mohamed Gouda, Professor Simon Lam, Professor Miro Malek, and Professor Lou Rosier, the members of my committee, for their guidance in this endeavor.

I am grateful to Doug Stuart for countless hours of discussion on the formalism presented in this work. I wish to thank Prasanna Amerasinghe, Raymond Lee, and Tom Porter for their helpful suggestions. Finally, I wish to thank Tris Casey (soon to be Mrs. Jahanian), my brother Farzan, and my parents for their encouragement and enthusiasm all along.

Farnam Jahanian
The University of Texas at Austin
June 1988

ABSTRACT

This dissertation proposes a formalism for the specification and verification of timing properties of real-time systems. Reasoning about properties of a real-time system requires one to consider both relative and absolute timing of events. Relative timing concerns the order in which events occur, such as mutual exclusion and precedence constraint properties. Absolute timing concerns the stringent timing restrictions imposed on a system, such as a response time deadline or a minimum elapsed time between occurrences of two events.

The approach is based on Real Time Logic (RTL), a logic invented primarily for the specification of both relative and absolute timing of events. The notion of an event occurrence is central to RTL; an event occurrence marks a point in time which is of significance to the behavior of a system. Hence, concurrency is modeled as a partial ordering of the event occurrences in the system. A system specification and a property to be verified can be expressed as arithmetical relations on algebraic expressions involving the event occurrences. To verify the property with respect to the system specification, we prove that the property is a theorem derivable from the specification. Relationship of RTL to Presburger Arithmetic is discussed and a verification technique based on inequality provers is explored.

The dissertation also introduces a specification language, Modechart, for real-time systems. The semantics of Modechart is described in terms of RTL formulas. In Modechart, we make use of the concept of *modes* which can be thought of as partitioning the state space of a system. Intuitively, modes can be viewed as control information that impose structure on the operation of a system. Modes are arranged hierarchically. Furthermore, modes at the same level of hierarchy can be related in one of two ways: in series or in parallel. A transition can be specified between two modes in series, but no transition is allowed between modes in parallel. The language allows sporadic/periodic actions in modes as well as constructs for specifying timing constraints such as delays and deadlines on mode transitions. Verification procedures are introduced for showing a Modechart specification satisfies a property expressed as an RTL formula.

Table of Contents

Chapter 1

Introduction

With the rapid increase in the use of computers in real-time applications, e.g., industrial plant control, avionic systems, and patient-monitoring instruments, the need for a formal approach to the specification and verification of these systems has become particularly important. For these applications, computers are used to monitor and control potentially unsafe processes where a run-time failure may result in catastrophic destruction of life and property. The difficulty in designing and analyzing the software systems for these applications is compounded by two important characteristics of the real-time environment, namely, the requirement to continually satisfy stringent timing constraints, and the need to guard against an imperfect execution environment.

This report proposes a formalism for the specification and analysis of timing properties in hard real-time systems. The primary goal of the proposed approach is to describe and verify the design of systems for which the absolute timing of events in addition to their relative ordering is important. Given a system specification and a timing property under investigation, correctness of a design is established by showing the timing property is consistent with the system specification. The same mathematical notations are used in describing the system specifications and the properties under investigation, thus the verification can be carried out as proving theorems.

Although there has been wide interest among researchers in describing and verifying concurrent systems, most of the work in this area does not consider the real-time constraints that may be imposed on the behavior of systems. Since timing constraints are crucial to the correct operation of systems in a real-time environment, description and verification of properties of these systems must consider the temporal restrictions imposed by the timing constraints. The work

proposed in this report will focus on this issue. In particular, the proposed formalism is characterized by the following three attributes: First, it provides a uniform way for the specification and analysis of both relative and absolute timing of events. Second, it puts much emphasis on the mechanical verification of properties under investigation. Third, the formalism is intended mainly for analyzing system specifications instead of detailed program designs. This is important for keeping the analysis manageable for mechanization, since automated analysis of large finished designs is not likely to be practical.

1.1. Timing Constraints: Two Examples

The specification problem for real-time systems is more complex since the absolute timing behavior (the timing of events measured by a real-time clock) and not only the functional behavior of a system is important. Consequently, reasoning about properties of a real-time system requires one to consider both relative and absolute timing of events. Relative timing concerns the relative order in which events occur. Properties such as mutual exclusion and precedence constraints fall within this category. Absolute timing concerns the stringent timing restrictions imposed on a system. Response time deadline or minimum elapse time between occurrences of two events are examples of absolute timing properties.

We present two examples to illustrate the potential difficulties in the specification and verification of time-critical systems.

Example 1:

Consider a computer on board a train which is going into a railway crossing. The train is to stop if, after 45 seconds, the controller at the crossing fails to lower the gate. The following piece of Ada code might be used for this purpose. (The use of the **timed entry call** below follows the suggestion in the Ada Language Reference Manual [Ada Manual 83].)

```
select
  LOWER_GATE.REQUEST;
or
  delay 45.0;
  STOP_TRAIN;   ---controller not responding, stop the train
end select;
```

The **select** statement above has two alternatives. The first alternative is a *rendezvous* with the gate controller (the entry call LOWER_GATE.REQUEST). The other alternative simultaneously starts a watchdog timer. The semantics of the *timed entry call* is given in the Ada Language Reference manual: "If a *rendezvous* can be started within the specified duration (or immediately, as for a conditional entry call, for a negative or zero delay), it is performed and the optional sequence of statements after the entry call is then executed. Otherwise, the entry call is canceled when the specified duration has expired and the optional sequence of statements of the delay alternative is executed." Thus if the controller does not respond within 45 seconds, the train will automatically be stopped. However, if the *rendezvous* with the controller starts within the specified time but takes a long time to complete (or it never completes due to a controller breakdown in the middle of the *rendezvous*), then the watchdog timer will not be able to take effect according to the Ada Language Reference Manual. The train will therefore not stop even though the intended timing constraint is to stop the train when the controller fails to lower the gate within 45 seconds.†

The above example illustrates two points. First, a timing constraint may involve the execution of more than one action (task) in a distributed environment. Since different actions may synchronize and interact in many ways, the specification model must allow a system designer to express at least the timing constraints which are of practical importance. Second, execution of an action, e.g., lowering of a gate, should not be viewed as a single event. To specify the

† There is no easy way to express timing constraints like this in Ada. In [Mok 88], an annotation system, based on the logic proposed in this thesis, is introduced which provides a precise way to specify the intended timing property.

timing behavior of real-time systems, it is crucial to model the execution of an action by two events: one marking the initiation of an action and the other denoting the completion of the action. In the above example, there would be no ambiguity if one could specify the intended timing constraint as *the train should stop if the event marking the completion of the rendezvous does not happen in bounded time*.

Example 2:

Suppose an external event, e.g., pressing of a push button, invokes two concurrent processes P_1 and P_2 to update two input sensors X and Y, respectively. We specify a timing constraint for each process which requires the sensors to be updated within 1 time unit. The global state of the system can be represented by the vector (x,y). The two variables x and y denoting the values of the input sensors are initially zero. Two different computations for the system in the interleaving model of concurrency are shown in Figure 1.1. It is unclear what time value should be assigned to the second (intermediate) state in either computation. There are, of course, two choices: t or $t+1$. If time=$t+1$ is associated with the intermediate states, then it is unclear what the values of x and y are at time $t+1$. (Similar ambiguity arises if time=t is associated with the intermediate states).

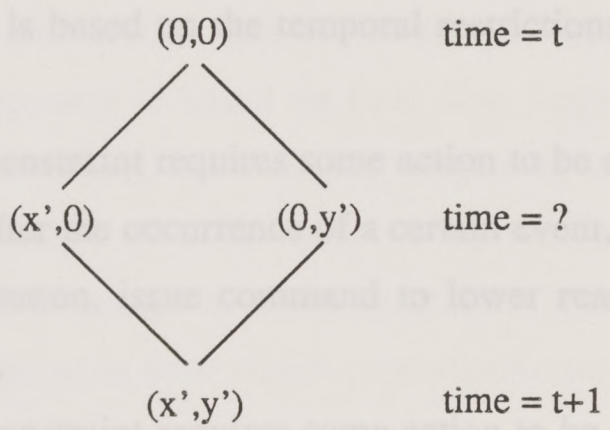


Figure 1.1

The point of the above example is that viewing a behavior of a system as a total ordering of events may generate system states which may not reflect the behavior intended by the specification. Furthermore, if a computation of a system is defined as a sequence of states, the notion of a system state at a point in time is not clear because the system state may be undergoing a transition right at that point and the value of a variable may be undefined when it is changing. There is, however, no ambiguity regarding a change in the value of a variable at an instant of time because it refers to the occurrence of an event at a point in time. The formalism proposed in this thesis models concurrency as a partial ordering of events in the system. Hence, the notion of a *system state* is non-existent. A system behavior is defined as the occurrences of events which are partially ordered in time. We shall return to this issue in Chapter 2.

1.2. Classifying Timing Constraints

Since timing constraints are assertions specifying performance requirements of a system, they are crucial to the correct operation of systems in a real-time environment. This section presents an overview of two classifications of timing constraints in the hard real-time environment.

The first classification, often used in formulating scheduling problems, catagorizes the timing constraints into two types: sporadic and periodic [Mok 1983]. This classification is based on the temporal restrictions imposed on the execution of actions.

- A *sporadic* timing constraint requires some action to be executed before a specified deadline after the occurrence of a certain event, e.g., upon pressing the down-pushbutton, issue command to lower reactor control rods within 10 time units.
- A *periodic* timing constraint requires some action to be executed at fixed intervals when some condition is true, e.g., while the plane is in the *emergency landing mode*, sample velocity and altitude every 1 second.

The second classification catagorizes timing constraints by three types of temporal restrictions on the events in a system [Dasarathy 85]. These restrictions may be imposed on the behavior of a system or its users.

- A *maximum* requirement imposes an upper limit on the amount of time that may elapse between the occurrences of two events, e.g., the system responds within 200 time units after the switch is turned to the ON position.
- A *minimum* requirement imposes a lower limit on the amout of time that may elapse between two events, e.g., if the computation time of an action is 20 time units, it will stop at least 20 time units after it starts.
- A *durational* requirement imposes a restriction on the length of time that a signal (stimulus or response) must last, e.g., press pushbutton 0 for at least 3 seconds to access the operator.

The formalism introduced in this thesis provides a uniform way for specification of both relative and absolute timing of events including the above timing constraints regardless of the classification preferred by a system designer.

1.3. Overview of the Dissertation

Having discussed the motivation for this work and two classifications of timing constraints in a hard real-time enviornment, the subsequent chapters propose a formalism for the specification and verification of timign properties of real-time systems. The approach is based on Real time Logic (RTL), a logic especially amenable for reasoning about timing properties of these systems. Chapter 2 presents an overview of RTL followed by the formal syntax and semantics of the logic. The notion of an event occurrence is central to RTL; an event occurrence marks a point in time which is of significance to the behavior of a system. Hence, concurrency is modeled as a partial ordering of the event occurrences in the system. A system specification and a property to be verified can be expressed as *occurrence* relations on the set of event instances, and as

arithmetical relations on algebraic expression involving the time of occurrence of instances of events. To verify the property with respect to the system specification, we prove that the property is a theorem derivable from the specification. The chapter also provides an example illustraing a RTL specification of a system responsible for monitoring the control rods in a nuclear reactor. A safety assertion about the system is verified by employing the first-order logic inference rules.

Chapter 3 presents an alternative notation for RTL in which a system specification and a property to be verified are expressed as artithmetical relations on algebraic expressions involving event occurrences. The relationship of RTL formulas to Presburger arithmetic is exploited and a verification technique based on inequality provers is discussed. The chapter concludes with a discussion on the undecidability of RTL.

Chapter 4 introduces a graph-theoretic approach for the verification of a class of timing properties in real-time systems which are expressible in a subset of Real Time Logic formulas. The motivation for this approach is that the RTL formulas describing real-time system in many cases consist of arithmetic inequalities (which may be quantified) involving two functions and an integer constant. The proposed technique exploits this class of formulas to obtain a more practical verification procedure. Mechanical verification of the safety assertion in the nuclear reactor example is shown using the approach described in this chapter.

Chapter 5 introduces a specification language, Modechart, for real-time systems. The semantics of Modechart is described in terms of RTL formulas. In Modechart, we make use of the concept of *modes* which can be thought of as partitioning the state space of a system. Intuitively, modes can be viewed as control information that impose structure on the operation of a system. Modes are arranged hierarchically. Furthermore, modes at the same level of hierarchy can be related in one of two ways: in series or in parallel. A transition can be

specified between two modes in series, but no transition is allowed between modes in parallel. In addition to allowing sporadic/periodic actions in modes, the language also provides the constructs for specifying timing constraints such as delays and deadlines on mode transitions.

Chapter 6 is on the verification of systems specified in Modechart. Given a Modechart specification and a timing property expressed as an RTL formula, the goal is to verify that every computation of the system satisfies the desired timing property. Chapter 6 defines the notion of a system computation and it presents an algorithm for constructing a transition graph which represents the computations of a given Modechart system. The chapter also introduces two classes of timing properties expressed in RTL. For each class, decision procedures are described for proving whether a given transition graph satisfies a property in the class.

Chapter 7 is the conclusion and presents some thoughts for future research.

Chapter 2

Real Time Logic

2.1. Introduction

This chapter introduces a formal language, Real Time Logic, especially amenable for reasoning about timing properties of real-time system. Real Time Logic (RTL) is invented to describe systems for which the absolute timing of events and not only their relative ordering is important. RTL is a first-order theory and has no modal operators. Although the analysis of real-time systems necessarily involves reasoning about system behavior with respect to time, we do not think that temporal logic is appropriate for the task at hand. However, since a variety of temporal logics have been proposed by a number of researchers for reasoning about the temporal properties of real-time programs (e.g., [Bernstein & Harter 81], [Schwartz et al 83], [Ostroff 87]).

In general, temporal logic is more concerned with the relative order in which actions are executed rather than the absolute timing of events. Example 2 in the previous chapter discussed some of the potential problems with the interleaving model of concurrency and viewing a system behavior as a sequence of system states. Suppose a system consists of three actions A, B, and C such that B and C are executed in parallel after completing action A. We represent this system by the following notation

A; (B||C)

where ';' and '||' denote the sequential and parallel execution of actions, respectively. The computations of the above system is characterized in temporal logic by two execution sequences: ABC and ACB, but nothing is said about the completion time of C in the first sequence or the completion time of B in the second sequence. For our purposes, these absolute time values may appear in a safety

assertion and are therefore important. Even for the same sequence ABC, we may want to distinguish between the execution that schedules A at time=0, B at time=1 and C at time=2 from the execution that schedules A at time=0, B at time=2 and C at time=3. For temporal logic, this distinction is unimportant. Furthermore, it is at the least awkward in some cases to reason about system behavior in terms of execution sequences alone. For example, if all three actions A, B and C each takes 1 time unit to execute and the composite action above must be completed by time=2, then B and C must be executed in parallel on two processors. In this case, neither of the sequences ABC nor ACB captures the desired behavior.

It has been suggested by some researchers (e.g., [Lamport 83]) that real time can be modelled in temporal logic simply as another global variable: the clock. The assertions involving real time will simply be temporal logic formulas involving the clock variable. The main problem with this approach is when to increment the clock variable in relation to the other activities in the system. One possible way to achieve this is to insert the assignment statement:

clock := clock+c

at the end of every action where c is the time required to execute the corresponding action. If all actions are executed in some sequential order, such as on a single processor, then the clock variable will indeed keep track of real time. (For this to work, we should treat each action and the clock update statement at its end to be one inseparable atomic action.) Unfortunately, this is not true when two or more actions can be executed in parallel.

Another way to model real time in temporal logic is to create a process whose body is an infinite loop which increments the clock variable *ad infinitum* ([Ostroff & Wonham 87]. This approach, however, only begs the question since we must then find appropriate scheduling restrictions so as to meter the progress of the clock process in relation to the other processes in the system. Since most temporal logics dictate only minimal restrictions (usually a fairness

criterion) on process scheduling, imposing additional scheduling restrictions seems to move against the original spirit of using temporal logic for program verification. More importantly, these additional scheduling restrictions will depend on the execution environment, e.g., the rate of scheduling the clock process when there is one processor will differ when two processors are available to execute all the processes. Proof rules which are sound under one execution environment may not apply under a different execution environment.

The previous discussion brings out a fundamental issue in proving real-time properties of time-critical systems, namely, the validity of the assertions that we want to prove about these systems often cannot be established without knowing more details about the run-time scheduler. This is unlike the usual safety and liveness properties of non-time-critical systems which we want to hold true in spite of the scheduler as long as we are assured that some fairness criterion in scheduling is met. Our Real Time Logic promises to provide a uniform way to incorporate different scheduling disciplines in the inference mechanism.

2.2. Real Time Logic

The first subsection describes an informal overview of RTL. The subsequent two subsections provide the formal syntax and the semantics of the logic. The final subsection introduces a convenient notation in RTL for describing a property of a system over a time interval.

2.2.1. Overview

Real Time Logic is a first order predicate logic invented primarily for reasoning about timing properties of real-time systems. It provides a uniform way for the specification of both relative and absolute timing of events. In RTL, we reason about individual occurrences of events where an event occurrence marks

a point in time which is of significance to the behavior of the system. There is an important distinction between an action and an event in RTL. An action is an operation which requires a non-zero but bounded amount of system resources. However, events serve only as temporal markers. An occurrence of an event defines a time value, namely its time of occurrence, and imposes no requirement on system resources. The execution of an action is represented by two events: one denoting its initiation and the other denoting its completion.

Events have unique names. Two classes of events are of particular interest: (1) start/stop events marking the initiation and completion of an action, and (2) transition events denoting a change in a state variable.

Start and Stop Events: We use the notation ↑A to represent the event marking the initiation of action A, and ↓A to denote the event marking the completion of action A. For instance, ↑SAMPLE and ↓SAMPLE represent the events corresponding to the start and the stop of action SAMPLE, respectively.

Transition Events: A state variable may describe a physical aspect or a certain property of a system, e.g., an autopilot switch which is either ON or OFF. The execution of an action may cause the value of one or more state variables to change. A state attribute, S is a predicate which asserts that a state variable takes on a certain value in its domain. For example, S may denote the predicate: the autopilot is ON. The corresponding state variable transition events, represented syntactically by (S:=T) and (S:=F), denote respectively the events that mark the turning on and off of the autopilot switch. Whenever it is unambiguous, we shall use state variable and state attribute interchangeably.

In addition to the above two classes, other classes of events can be specified when modeling a real time system. For instance, the class of external events can be introduced to denote the events that cannot be caused to happen by the computer system but can impact a system behavior. We use the notation of any name in capital letters prefixed by the special letter Ω (Omega) to denote an external event. For example, ΩBUTTON1 represents the external event

associated with pressing button 1.

The *occurrence relation*, denoted by the letter Θ (Theta), is introduced to capture the notion of real time. The event constants introduced earlier represent the things that can happen in a system. The occurrence relation assigns a time value to each occurrence of an event which happens. Informally, $\Theta(e,i,t)$ denotes that the ith occurrence of an event e happens at time t, where e is an event constant, i is a positive integer term, and t is non-negative integer term. For instance, $\Theta(\downarrow\text{SAMPLE},1,x)$ denotes that the first occurrence of the event marking the completion of action SAMPLE happens at time x.

The notion of an occurrence relation is central to RTL. In particular, a specification of a system and the timing requirements on its behavior are restrictions on the occurrence relation and its arguments. Observe that the occurrence relation does not require that all occurrence of an event must happen since an event may occur only a finite number of times or even not at all.

RTL predicates are formed from the occurrence relation, or from the mathematical relations ($=, <, \leq, >, \geq$) and algebraic expressions allowing integer constants, variables, addition, and multiplication by constants. RTL formulas are constructed using the occurrence relation, the equality/inequality predicates, universal and existential quantifiers, and the first-order logical connectives $(\neg,\wedge,\vee,\rightarrow)$[†]. Before presenting a formal treatment of Real time Logic in the next section, a simple example will be shown to illustrate how the specification of a system can be expressed in RTL.

Example:

Consider the English description of a simple system which samples and displays information on demand from an external stimulus. Upon pressing button #1, action SAMPLE is executed within 30 time units. During each execution of this action, the information is sampled and subsequently transmitted to the display

† The standard precedence order is assumed for these connectives. $\neg$ has the highest precedence, $\wedge$ and $\vee$ the next highest precedence, and $\rightarrow$ the lowest precedence.

panel. The computation time of action SAMPLE is 20 time units. The following set of formulas is a partial description of the system in RTL:

$$\forall i \forall t\ \Theta(\Omega BUTTON1,i,t) \rightarrow [\exists\!\!\!\!-\, x,y\ \Theta(\uparrow SAMPLE,i,x) \wedge \Theta(\downarrow SAMPLE,i,y) \wedge t \leq x \wedge y \leq t + 30\ ]$$

$$\forall i \forall x,y\ [\ \Theta(\uparrow SAMPLE,i,x) \wedge \Theta(\downarrow SAMPLE,i,y)\ ] \rightarrow x + 20 \leq y$$

Recall that a transition event marks a change in the value of a state variable. RTL provides a notational device, called a state predicate, for asserting the truth value of a state variable (e.g., the autopilot switch of an airplane being in the ON position) during an interval. The syntax and formal definition of state predicates are presented at the end of section 2.

2.2.2. The Language of Real Time Logic

We begin by introducing the language of Real Time Logic. The formulae of RTL are made up of the following symbols:

- The truth symbols *true* and *false*
- A set of variable symbols
- A set of constant symbols **C**
- A set of event constant symbols **D**
- The function symbol '+'
- The predicate symbols $<, \leq, >, \geq, =$
- The occurrence relation symbol Θ
- The logical connectives $\wedge, \vee, \neg$ and $\rightarrow$.
- Existential and universal quantifier symbols $\exists\!\!\!\!-$, $\forall$.

The *terms* of RTL are expressions built up according to the following rules:

- The constant symbols in **C** are terms.
- The variable symbols are terms.
- If t_1 and t_2 are terms, then the function application

$$t_1 + t_2$$

is a term.

The *propositions* of RTL are constructed according to the following rules:

- The truth symbols *true* and *false* are propositions.
- If t_1 and t_2 are terms and ρ is an inequality/equality predicate symbol, then

$$t_1 \rho t_2$$

is a proposition.

- If t_1 and t_2 are terms and e is an event constant, then

$$\Theta(e\ , t_1\ , t_2)$$

is a proposition.

The *formulae* of RTL are constructed from the propositions, logical connectives and quantifiers in the usual fashion.

2.2.3. The Meaning of an RTL Formula

An interpretation for an RTL formula must assign a meaning to each of the free symbols in the formula. It will assign elements from a domain to the constants, functions (over the domain) to the function symbols, and relations (over the domain) to the predicate symbols.

Let **N** be the set of natural numbers, and **E** be a set of events. An interpretation I over the domain $\mathbf{N} \cup \mathbf{E}$ assigns values to each of a set of constant, function, and predicate symbols, as follows:

Each element in the set of constant symbols **C** is assigned an element in **N**. Each element in the set of event constant symbols **D** is assigned an element in **E**. The function symbol '+' is integer addition. The predicate symbols $<$, $\leq$, $>$, $\geq$, and $=$ are assigned the usual equality/inequality binary relations. The predicate symbol Θ is assigned an *occurrence relation.*

Definition: An *occurrence relation* is any relation on the set

$$\mathbf{E} \times \mathbf{Z}^{+} \times \mathbf{N}$$

where **E** is a set of events, $\mathbf{Z}^{+}$ is the set of positive integers, and **N** is the set of natural numbers, such that the following axioms hold:

Monotinicity Axioms: For each event e in the set **E**,

$$\forall i \forall t \forall t' \; [\, \Theta(e,i,t) \wedge \Theta(e,i,t') \,] \rightarrow t = t'$$

$$\forall i \forall t \; [\, \Theta(e,i,t) \wedge i > 1 \,] \rightarrow [\, \exists\!\!\!-\, t' \; \Theta(e,i\text{-}1,t') \wedge t' < t \,]$$

The first axiom requires that at most one time value can be associated with each occurrence i of an event e, i.e., the same occurrence of an event cannot happen at two distinct times. The second axiom expresses the requirement that if the ith occurrence of an event e happens, then the previous occurrences of e must have happened earlier. This axiom also requires that two distinct occurrences of the same event must happen at different times.

Start/Stop Event Axioms: For each pair of start/stop events in the set **E**,

$$\forall i \forall t \; \Theta(\downarrow A,i,t) \rightarrow [\, \exists\!\!\!-\, t' \; \Theta(\uparrow A,i,t') \wedge t' < t \,]$$

where $\uparrow A$ and $\downarrow A$ denote the events marking the start and stop of an action A, respectively. The above axiom requires every occurrence of a stop event to be preceded by a corresponding start event.

Transition Event Axioms: For the transition events in the set **E** corresponding to a state variable S,

if S is initially true,

$$\Theta((S:=T),1,0)$$

$$\forall i \forall t \; \Theta((S:=F),i,t) \rightarrow [\, \exists\!\!\!-\, t' \; \Theta((S:=T),i,t') \wedge t' < t \,]$$

$$\forall i \forall t \; \Theta((S:=T),i+1,t) \rightarrow [\, \exists\!\!\!-\, t' \; \Theta((S:=F),i,t') \wedge t' < t \,]$$

if S is initially false,

$$\Theta((S:=F),1,0)$$
$$\forall i \forall t \ \Theta((S:=T),i,t) \rightarrow [\exists\!\!\!-\, t' \ \Theta((S:=F),i,t') \wedge t' < t\,]$$
$$\forall i \forall t \ \Theta((S:=F),i+1,t) \rightarrow [\exists\!\!\!-\, t' \ \Theta((S:=T),i,t') \wedge t' < t\,]$$

The preceding transition event axioms define the order in which two complementary transition events can occur depending on whether S is initially true or false.

2.2.4. State Predicates

A specification of a real-time system often refers to properties of the system over time. One way to express these assertions is to introduce predicates which are time-dependent, as is done in temporal logic. In RTL, these assertions are expressed as formulae involving the occurrence relation on the appropriate transition events and inequality relations on the time of occurrences of these events. For convenience, RTL uses a notational device, called a state predicate, for asserting the truth value of a state variable (e.g., the autopilot switch of an airplane being in the ON position) during an interval.

Suppose S is a state variable whose truth value remains unchanged over an interval. RTL provides nine different forms of state predicates to assert the value of S over an interval, depending on the boundary conditions:

S[x,y], S(x,y), S<x,y>, S[x,y), S[x,y>, S(x,y], S(x,y>, S<x,y], and S<x,y).

Each state predicate qualifies the timing of two events, one marking the transition event that changes the value of the state variable to true and the other marking the transition event that changes the value of S to false. The two arguments, x and y, in the state predicates are used in conjunction with the symbols "[", "]", "(", ")", "<" and ">" to denote an interval over which the state variable remains true.

The convention we use for arriving at this syntax requires some explanation. Suppose E_t and E_f denote the transition events making S true and false, respectively. Informally,

"[x " denotes that E_t occurs at time x,
"(x " denotes that E_t occurs before or at time x,
"<x " denotes that E_t occurs before time x,
" y]" denotes that E_f occurs at time y,
" y)" denotes that E_f does not occur before time y,
" y>" denotes that E_f does not occur before or at time y,

For example, the state predicate S[x,y] indicates that a state variable S is true exactly during the interval between x and y. That is, a transition event making S true occurs at time x; S remains true between x and y; and the transition event making S false occurs at time y. Consider another example, the state predicate S(x,y) denotes that a state variable is true during the interval between x and y, but it says nothing about the value of S before time x or after time y. Observe that we use this notation only when $x \leq y$ and that we do not require the second event to occur in all cases. The formal definitions of these state predicates follow.

Definition:

If a state variable S is initially true, $\Theta((S:=T),1,0)$,

$$S[x,y] \equiv \exists i \; \Theta((S:=T),i,x) \wedge \Theta((S:=F),i,y)$$
$$S[x,y) \equiv \exists i \; \Theta((S:=T),i,x) \wedge [\forall t \; \Theta((S:=F),i,t) \rightarrow y \leq t]$$
$$S[x,y> \equiv \exists i \; \Theta((S:=T),i,x) \wedge [\forall t \; \Theta((S:=F),i,t) \rightarrow y < t]$$
$$S(x,y) \equiv \exists i \exists t \; \Theta((S:=T),i,t) \wedge t \leq x \wedge [\forall t' \; \Theta((S:=F),i,t') \rightarrow y \leq t']$$
$$S(x,y] \equiv \exists i \exists t \; \Theta((S:=T),i,t) \wedge t \leq x \wedge \Theta((S:=F),i,y)$$
$$S(x,y> \equiv \exists i \exists t \; \Theta((S:=T),i,t) \wedge t \leq x \wedge [\forall t' \; \Theta((S:=F),i,t') \rightarrow y < t']$$
$$S<x,y> \equiv \exists i \exists t \; \Theta((S:=T),i,t) \wedge t < x \wedge [\forall t' \; \Theta((S:=F),i,t') \rightarrow y < t']$$
$$S<x,y] \equiv \exists i \exists t \; \Theta((S:=T),i,t) \wedge t < x \wedge \Theta((S:=F),i,y)$$
$$S<x,y) \equiv \exists i \exists t \; \Theta((S:=T),i,t) \wedge t < x \wedge [\forall t' \; \Theta((S:=F),i,t') \rightarrow y \leq t']$$

If a state variable S is initially false, $\Theta((S:=F),1,0)$,

$$S[x,y] \equiv \exists i\ \Theta((S:=T),i,x) \wedge \Theta((S:=F),i+1,y)$$
$$S[x,y) \equiv \exists i\ \Theta((S:=T),i,x) \wedge [\forall t\ \Theta((S:=F),i+1,t) \rightarrow y \leq t\ ]$$
$$S[x,y> \equiv \exists i\ \Theta((S:=T),i,x) \wedge [\forall t\ \Theta((S:=F),i+1,t) \rightarrow y < t\ ]$$
$$S(x,y) \equiv \exists i \exists t\ \Theta((S:=T),i,t) \wedge t \leq x \wedge [\forall t'\ \Theta((S:=F),i+1,t') \rightarrow y \leq t'\ ]$$
$$S(x,y] \equiv \exists i \exists t\ \Theta((S:=T),i,t) \wedge t \leq x \wedge \Theta((S:=F),i+1,y)$$
$$S(x,y> \equiv \exists i \exists t\ \Theta((S:=T),i,t) \wedge t \leq x \wedge [\forall t'\ \Theta((S:=F),i+1,t') \rightarrow y < t'\ ]$$
$$S<x,y> \equiv \exists i \exists t\ \Theta((S:=T),i,t) \wedge t < x \wedge [\forall t'\ \Theta((S:=F),i+1,t') \rightarrow y < t'\ ]$$
$$S<x,y] \equiv \exists i \exists t\ \Theta((S:=T),i,t) \wedge t < x \wedge \Theta((S:=F),i,y)$$
$$S<x,y) \equiv \exists i \exists t\ \Theta((S:=T),i,t) \wedge t < x \wedge [\forall t'\ \Theta((S:=F),i+1,t') \rightarrow y \leq t'\ ]$$

When both arguments of a state predicate are the same, two very useful predicates are defined. Specifically, the state predicate S(x,x) says that a system attribute S is true at an interval around time x. Similarly, S<x,x) denotes that S is true prior to time x, i.e., S becomes true sometime before time x and it remains true at least up to time x.

Similar state predicates can be defined for the case when a state variable is false during an interval: $\bar{S}[x,y]$, $\bar{S}(x,y)$, $\bar{S}<x,y>$, etc. The formal definitions for these predicates are straightforward: replace each (S:=T) with (S:=F) and vice versa in the above definitions.

2.3. Example: Moving Control Rods in a Reactor

This section describes an example illustrating the specification of a system in RTL and it verifies a safety property with respect to the specification.

2.3.1. Specification

This example involves a system used to monitor and control a reactor. Specifically, it concerns the part of the system responsible for moving the control rods in a reactor. (Lowering a control rod slows down a nuclear reaction.) For simplicity, we assume that the system is currently configured with two primary control rods. As shown in Figure 2.1, three asynchronous components are involved in moving the control rods: Subsystem 1, Subsystem 2, and Manager.

Upon pressing pushbutton #1, Subsystem 1 executes an action to ensure the conditions are satisfied for moving control rod #1, then it requests permission from the manager to move a rod. After the manager grants the request, Subsystem 1 issues the appropriate commands to the reactor to move rod #1. The specification imposes two timing constraints on Subsystem 1:

- When the request is granted by the manager, the action to move the rod must be started within 5 time units, and
- at most 20 additional time units is required to complete the action to move the rod.

Subsystem 2 performs similar tasks for moving control rod #2 with identical timing constraints. The details about the behavior of Manager are hidden at this level of specification. However, the following timing constraint is imposed on Manager:

- After granting a request, Manager must wait at least 30 time units before granting another request.

There is no signal from Subsystem 1 or Subsystem 2 to inform Manager after a rod movement has been completed. Hence, Manager can only rely on the enforcement of the preceding timing constraint to decide when a granted request has been released, and then grant permission to move a rod to a waiting subsystem.

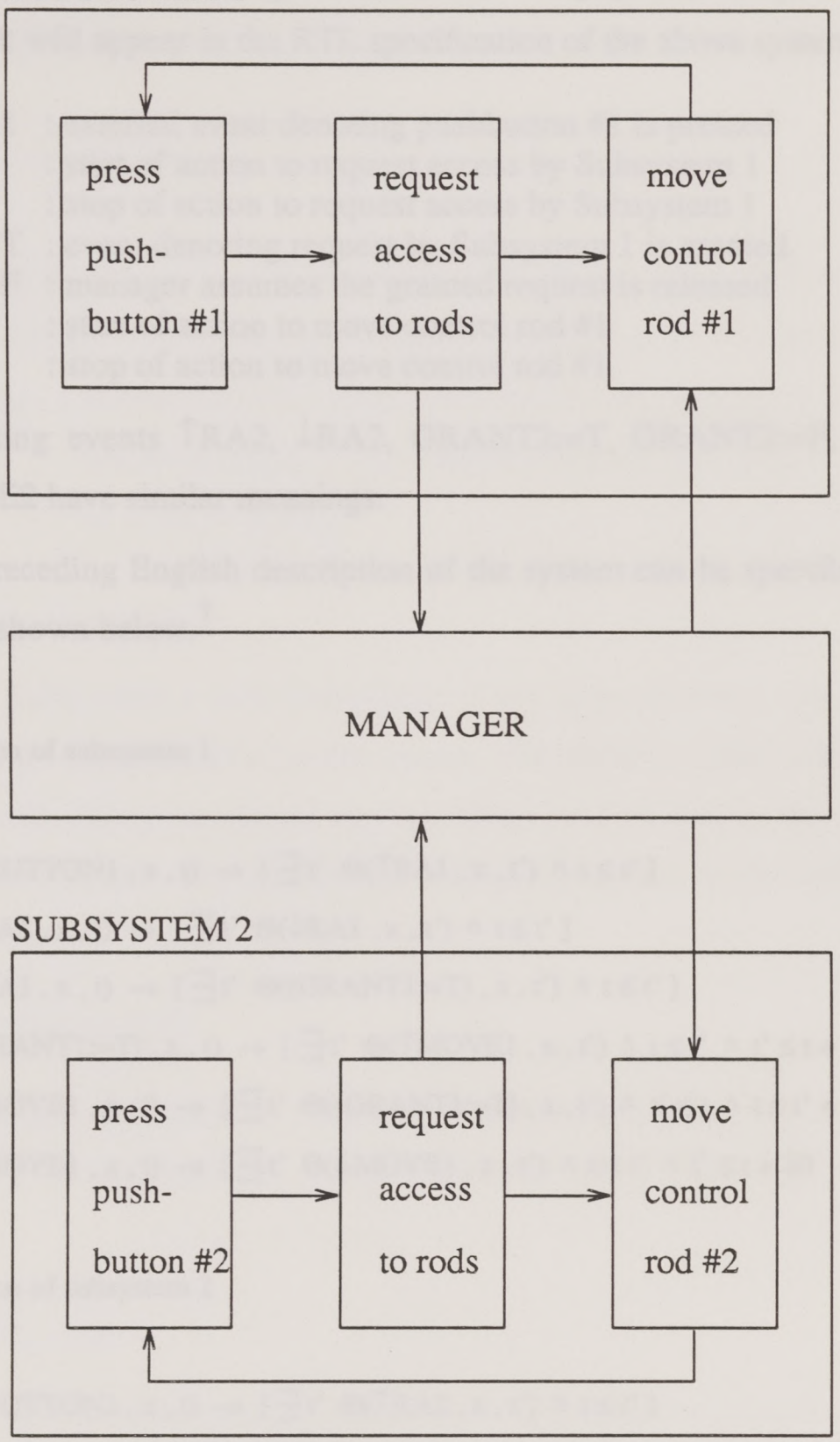

SUBSYSTEM 1
press push-button #1
request access to rods
move control rod #1
MANAGER
SUBSYSTEM 2
press push-button #2
request access to rods
move control rod #2

FIGURE 2.1

For ease of understanding, we list below the physical interpretation of the events which will appear in the RTL specification of the above system.

ΩBUTTON1 : external event denoting pushbutton #1 is pressed
↑RA1 : start of action to request access by Subsystem 1
↓RA1 : stop of action to request access by Subsystem 1
GRANT1:=T : event denoting request by Subsystem 1 is granted
GRANT1:=F : manager assumes the granted request is released
↑MOVE1 : start of action to move control rod #1
↓MOVE1 : stop of action to move control rod #1

The remaining events ↑RA2, ↓RA2, GRANT2:=T, GRANT2:=F, ↑MOVE2, and ↓MOVE2 have similar meanings.

The preceding English description of the system can be specified formally in RTL, as shown below.†

#

specification of subsystem 1

#

$\forall x \forall t\ \Theta(\Omega\text{BUTTON1}, x, t) \rightarrow [\exists t'\ \Theta(\uparrow\text{RA1}, x, t') \wedge t \leq t']$ (SP1)

$\forall x \forall t\ \Theta(\uparrow\text{RA1}, x, t) \rightarrow [\exists t'\ \Theta(\downarrow\text{RA1}, x, t') \wedge t \leq t']$ (SP2)

$\forall x \forall t\ \Theta(\downarrow\text{RA1}, x, t) \rightarrow [\exists t'\ \Theta((\text{GRANT1:=T}), x, t') \wedge t \leq t']$ (SP3)

$\forall x \forall t\ \Theta((\text{GRANT1:=T}), x, t) \rightarrow [\exists t'\ \Theta(\uparrow\text{MOVE1}, x, t') \wedge t \leq t' \wedge t' \leq t + 5]$ (SP4)

$\forall x \forall t\ \Theta(\uparrow\text{MOVE1}, x, t) \rightarrow [\exists t'\ \Theta((\text{GRANT1:=T}), x, t') \wedge t' \leq t \wedge t \leq t' + 5]$ (SP4')

$\forall x \forall t\ \Theta(\uparrow\text{MOVE1}, x, t) \rightarrow [\exists t'\ \Theta(\downarrow\text{MOVE1}, x, t') \wedge t < t' \wedge t' \leq t + 20$ (SP5)

#

specification of subsystem 2

#

$\forall x \forall t\ \Theta(\Omega\text{BUTTON2}, x, t) \rightarrow [\exists t'\ \Theta(\uparrow\text{RA2}, x, t') \wedge t \leq t']$ (SP6)

$\forall x \forall t\ \Theta(\uparrow\text{RA2}, x, t) \rightarrow [\exists t'\ \Theta(\downarrow\text{RA2}, x, t') \wedge t \leq t']$ (SP7)

$\forall x \forall t\ \Theta(\downarrow\text{RA2}, x, t) \rightarrow [\exists t'\ \Theta((\text{GRANT2:=T}), x, t') \wedge t \leq t']$ (SP8)

$\forall x \forall t\ \Theta((\text{GRANT2:=T}), x, t) \rightarrow [\exists t'\ \Theta(\uparrow\text{MOVE2}, x, t') \wedge t \leq t' \wedge t' \leq t + 5]$ (SP9)

---

† Certain RTL axioms, as described in [Jahanian & Mok 86a], are omitted here since they do not play a role in the safety analysis of this example.

$$\forall x \forall t\ \Theta(\uparrow \text{MOVE2}, x, t) \rightarrow [\exists t'\ \Theta((\text{GRANT2:=T}), x, t') \wedge t' \leq t \wedge t \leq t' + 5] \quad \text{(SP9')}$$

$$\forall x \forall t\ \Theta(\uparrow \text{MOVE2}, x, t) \rightarrow [\exists t'\ \Theta(\downarrow \text{MOVE2}, x, t') \wedge t < t' \wedge t' \leq t + 20 \quad \text{(SP10)}$$

#

specification of manager

#

$$\forall x \forall t\ \Theta((\text{GRANT1:=T}), x, t) \rightarrow [\exists t'\ \Theta((\text{GRANT1:=F}), x+1, t') \wedge t + 30 \leq t'] \quad \text{(SP11)}$$

$$\forall x \forall t\ \Theta((\text{GRANT2:=T}), x, t) \rightarrow [\exists t'\ \Theta((\text{GRANT2:=F}), x+1, t') \wedge t + 30 \leq t'] \quad \text{(SP12)}$$

$$\forall x,y\ \forall t_1,t_2,t_3,t_4\ [\Theta((\text{GRANT1:=T}), x, t_1) \wedge \Theta((\text{GRANT1:=F}), x+1, t_2) \wedge$$
$$\Theta((\text{GRANT2:=T}), y, t_3) \wedge \Theta((\text{GRANT2:=F}), y+1, t_4)]$$
$$\rightarrow t_4 < t_1 \vee t_2 < t_3 \quad \text{(SP13)}$$

2.3.2. Safety Assertion

Since Subsystem 1 and Subsystem 2 are asynchronous, the requests to move the rods are made at arbitrary times. The desired safety property is that the reactor rods should be moved one at a time. In other words, execution of the action to move rod #1 may not overlap with the execution of the action to move rod #2.

Safety Assertion in RTL:

$$\forall i,j\ \forall t_1,t_2,t_3,t_4\ \Theta(\uparrow \text{MOVE1}, i, t_1) \wedge \Theta(\downarrow \text{MOVE1}, i, t_2) \wedge$$
$$\Theta(\uparrow \text{MOVE2}, j, t_3) \wedge \Theta(\downarrow \text{MOVE2}, j, t_4)$$
$$\rightarrow t_4 < t_1 \vee t_2 < t_3$$

2.3.3. Verification

The RTL formulas describing the system specification and the safety assertion were presented in subsections A and B. The proof of the safety assertion from the system specification is shown below.

1.	Show Safety Assertion		
2.	Show $\Theta(\uparrow\text{MOVE1}, \text{I}, T_1) \wedge \Theta(\downarrow\text{MOVE1}, \text{I}, T_2) \wedge$ $\Theta(\uparrow\text{MOVE2}, \text{J}, T_3) \wedge \Theta(\downarrow\text{MOVE2}, \text{J}, T_4)$ $\rightarrow T_4 < T_1 \vee T_2 < T_3$		
3.	a.	$\Theta(\uparrow\text{MOVE1}, \text{I}, T_1)$	ACP†
	b.	$\Theta(\downarrow\text{MOVE1}, \text{I}, T_2)$	ACP
	c.	$\Theta(\uparrow\text{MOVE2}, \text{J}, T_3)$	ACP
	d.	$\Theta(\downarrow\text{MOVE2}, \text{J}, T_4)$	ACP
	e.	Show $T_4 < T_1 \vee T_2 < T_3$	
4.	a.	$\exists$t' $\Theta((\text{GRANT1:=T}), \text{I}, \text{t'}) \wedge \text{t'} \leq T_1 \wedge T_1 \leq \text{t'} + 5$	$\forall$ Instantiation, 3.a, SP4', MP‡
	b.	$\Theta((\text{GRANT1:=T}), \text{I}, T_5) \wedge T_5 \leq T_1 \wedge T_1 \leq T_5 + 5$	$\exists$ Elimination, 4.a
5.	a.	$\exists$t' $\Theta((\text{GRANT1:=F}), \text{I+1}, \text{t'}) \wedge T_5 + 30 \leq \text{t'}$	$\forall$ Instantiation, 4.b, SP11, MP
	b.	$\Theta((\text{GRANT1:=F}), \text{I+1}, T_6) \wedge T_5 + 30 \leq T_6$	$\exists$ Elimination, 5.a
6.		$T_1 + 25 \leq T_6$	Transitivity Axiom, 4.b, 5.b
7.	a.	$\exists$t' $\Theta(\downarrow\text{MOVE1}, \text{I}, \text{t'}) \wedge \text{t'} \leq T_1 + 20$	$\forall$ Instantiation, 3.a, SP5, MP
	b.	$\Theta(\downarrow\text{MOVE1}, \text{I}, T_7) \wedge T_7 \leq T_1 + 20$	$\exists$ Elimination, 7.a
	c.	$T_2 \leq T_1 + 20$	Θ Relation Ax, 3.b, 7.b
8.		$T_2 + 5 \leq T_6$	Transitivity Axiom, 6, 7.c
9.	a.	$\Theta((\text{GRANT2:=T}), \text{I}, T_8) \wedge T_8 \leq T_3 \wedge T_3 \leq T_8 + 5$	similar to step 4.b
	b.	$\Theta((\text{GRANT2:=F}), \text{I+1}, T_9)$	similar to step 5.b
	c.	$T_4 + 5 \leq T_9$	similar to step 8
10.		$T_9 < T_5 \vee T_6 < T_8$	$\forall$ Inst, 4.b, 5.b, 9.a, 9.b, MP
11.		Show $T_6 < T_8 \rightarrow T_2 < T_3$	
		a. $T_6 < T_8$	ACP
		b. $T_6 < T_3$	Transitivity Axiom, 9.a, 11.a
		c. $T_2 + 5 < T_3$	Transitivity Axiom 11.b, 8
12.		$T_9 < T_5 \rightarrow T_4 < T_1$	Similar to proof in step 11
13.		$T_4 < T_1 \vee T_2 < T_3$	Disj Exploitation 10, 11, 12
14.		$\forall \text{i,j} \forall t_1, t_2, t_3, t_4$ $\Theta(\uparrow\text{MOVE1}, \text{i}, t_1) \wedge \Theta(\downarrow\text{MOVE1}, \text{i}, t_2) \wedge$ $\Theta(\uparrow\text{MOVE2}, \text{j}, t_3) \wedge \Theta(\downarrow\text{MOVE2}, \text{j}, t_4)$ $\rightarrow t_4 < t_1 \vee t_2 < t_3$	Universal Generalization

† Assumption for Conditional Proof

‡ Modus Ponens

Chapter 3

Inequality Provers and RTL

This chapter presents an alternative notation for RTL which first appeared in [Jahanian & Mok 86a]. Instead of using the occurrence relation Θ to relate event instances to their time of occurrence, the alternative notation defines a function which maps an instance of an event to a non-negative integer denoting its time of occurrence. This *occurrence* function, denoted by the "@" symbol, can be used in mathematical relations to specify the behavior of a system and the property to be verified. The second part of this chapter discusses the application of well-known inequality provers in mechanical verification of timing properties of systems specified in RTL.

3.1. Alternative RTL Notation: Occurrence Function

In chapter 1, an *occurrence relation* was defined to be any relation on the set

$$\mathbf{E} \times \mathbf{Z}^{+} \times \mathbf{N}$$

such that the monotonicity axioms, start/stop event axioms, and transition event axioms hold. An important property of the H relation is that an instance of an event e can be related to exactly one integer time value:

$$\forall i \forall t \forall t' \; [\, \Theta(e, i, t) \wedge \Theta(e, i, t') \,] \rightarrow t = t'$$

Consequently, one can define the following useful abbreviation. We define the function

$$@: \; \mathbf{E} \times \mathbf{Z}^{+} \rightarrow \mathbf{N}$$

such that @(e,i)=t if and only if Θ(e,i,t) e$\in$**E**, i$\in$$\mathbf{Z}^{+}$, and t$\in$**N**. The @ function is referred to as the *occurrence function*. Informally, @(e,i) is the time of the

i th occurrence of event e. For instance, @(↑SAMPLE,1) denotes the time of the first occurrence of the event marking the start of action SAMPLE.

RTL formulas are constructed from the equality/inequality predicates, state predicates, universal and existential quantifiers, and the first-order logic connectives. Relative and absolute timing requirements imposed by the system specification and the property under investigation are restrictions on the @ function. Unless otherwise stated in the subsequent chapters of this report, a reference to an RTL formula refers to the notation described above.

Example 1:

Consider the English description of a simple system which samples and displays information on demand from an external stimulus.

Specification: Upon pressing button #1, action SAMPLE is executed within 30 time units. During each execution of this action, the information is sampled and subsequently transmitted to the display panel. The computation time of action SAMPLE is 20 time units. The following set of formulas is a partial description of the subsystem in RTL.

$$\forall x \; @(\Omega BUTTON1,x) \leq @(\uparrow SAMPLE,x) \wedge @(\downarrow SAMPLE,x) \leq @(\Omega BUTTON1,x) + 30$$
$$\forall y \; @(\uparrow SAMPLE,y) + 20 \leq @(\downarrow SAMPLE,y)$$

Safety Assertion: If the transmitted information is displayed within 10 time units of the completion of action SAMPLE, we are assured that within 40 time units of pressing button #1, the requested information will be displayed. The safety assertion expressed in RTL follows.

$$\forall u \forall t \; @(\downarrow SAMPLE,u) \leq @(\Omega DISPLAY,t) \wedge @(\Omega DISPLAY,t) \leq @(\downarrow SAMPLE,u) + 10 \rightarrow$$
$$@(\Omega BUTTON1,u) < @(\Omega DISPLAY,t) \wedge @(\Omega DISPLAY,t) \leq @(\Omega BUTTON1,u) + 40$$

3.2. A Reasoning System Based on Inequality Provers

The major components of a verification system can be divided into three parts: system specification, property under investigation, and a verification procedure. A systems specification is a behavioral description of the real-time system, i.e., a set of constraints imposed on the behavior of the system in question. A property under investigation is an assertion (in RTL) which describes the property of the system to be verified. Finally, a verification procedure describes how to manipulate the known facts about the system to determine if the desired property is consistent with the specification.

It is crucial to point out that a property to be verified can be viewed as just another constraint added to the specification of the system. Thus we may employ the same model used in specifying a system to express the property in question. In other words, properties to be verified are expressed as RTL formulas. This allows the verification of a system to be carried out as proving a theorem: the property is shown to be a theorem derivable from the specification.

3.2.1. From RTL to Presburger Arithmetic

We have thus far shown how to express a system specification and a property to be verified in terms of RTL formulas. We now turn to the issue of reasoning about the timing properties of real-time systems. Even though there are more than one type of constants, a RTL formula currently consists of only algebraic relations and state predicates connected by first order logic operators. The state predicates in turn can be expanded into simple algebraic relations (see section 2.2.4). Hence the formulas that we have to analyze contain only integer terms. This raises the possibility of using a procedure similar to those used for deciding Presburger arithmetic. (Informally, Presburger formulas are those constructed from integer constants, integer variables, the addition function, the predicates ($<$, $\leq$, $>$, $\geq$, $=$), and the first-order logical connectives.) However, RTL formulas may contain constants other than integers (namely, event constants)

and these are not allowed in Presburger Arithmetic. We will show how RTL formulas can be mechanically transformed into the equivalent formulas in Presburger Arithmetic with uninterpreted functions.

Eliminating state predicates and non-integer constants from RTL formulas:

Given a system specification in terms of RTL formulas, the following transformation procedure describes how state predicates and non-integer constants can be eliminated.

1) In each formula, replace each state predicate with its definition as described in section 3.7. This step in effect substitutes, for each state predicate, a subformula containing only algebraic relations.

2) Recall that event constants only appear within occurrence functions of the form @(e,i) where e is an event constant and i is an integer variable/constant. For each event constant e, define an uninterpreted function f_e mapping from W to W where W is the set of non-negative integers. Then replace each appearance of @(e,i) in RTL assertions by the corresponding uninterpreted integer function $f_e(i)$.

Example 2:

The RTL formulas describing the specification and the safety assertion in Example 1 are represented by the following formulas in Presburger Arithmetic with uninterpreted functions.

Specification:

$$\forall x \; f(x) \le g_1(x) \wedge g_2(x) \le f(x) + 30$$
$$\forall y \; g_1(y) + 20 \le g_2(y)$$

Safety Assertion:

$$\forall u \, \forall t \; g_2(u) \le h(t) \wedge h(t) \le g_2(u) + 10 \rightarrow f(u) < h(t) \wedge h(t) \le f(u) + 40$$

In the above formulas, x, y, u, and t are variables, f, g_1, g_2, and h are

uninterpreted integer functions which correspond to the occurrence functions for pressing button #1 (event ΩBUTTON1), start of action SAMPLE (event ↑SAMPLE), stop of action SAMPLE (event ↓SAMPLE), and the external event denoting the display of information on the panel (event ΩDISPLAY), respectively.

3.2.2. Inequality Theorem Provers

Much research has been devoted to finding decision procedures for Presburger Arithmetic and determining the complexity of such algorithms, e.g., [Cooper 71], [Cooper 72]. Unfortunately, it has been shown that a decision procedure for the full Presburger Arithmetic is inherently computationally expensive and these algorithms are therefore not practical for nontrivial problems (see section 2 in [Shostak 79] for a brief historical perspective). Other works (e.g., [Bledsoe 74], [Shostak 77]) have concentrated on the subclass of quantifier-free Presburger formulas. [Shostak 79] further extended the work on quantifier-free Presburger logic to include uninterpreted functions. This allows us to have unrestricted universal and existential quantifiers in the formulas: The idea is to replace all the existentially quantified variables by the corresponding Skolem constants or functions. In [Bledsoe & Hines 80], Bledsoe and Hines describe a resolution-based procedure for proving theorems about general linear inequalities. These last two procedures provide the necessary tools for analysis of RTL formulas. As suggested earlier, applying the above transformation to the specification of a system expressed in RTL allows us to use the existing inequality theorem provers such as the one described in [Bledsoe & Hines 80].

Overview of Bledsoe & Hines' Approach:

To show that a safety assertion SA is consistent with a system specification SP, it must be proved that the formula F = SP ---> SA is valid. This is equivalent to showing that ¬F is unsatisfiable.

Bledsoe and Hines based their approach on a modified resolution procedure to show that ¬F is unsatisfiable. The formula ¬F is transformed into clausal form first. A special clause called TY is defined which is essentially a conjunction of ground inequality literals. (Due to a splitting procedure, ground literals are guaranteed to occur only in unit clauses.) The TY clause can be checked for a contradiction by using a ground inequality prover, such as the one described in [Shostak 77], [Shostak 79].

During each resolution cycle, a chaining procedure† is applied to generate a new resolvent R. If R is false, the proof is successfully terminated. If R is a ground inequality clause, it is added to TY, and then checked for a contradiction. Otherwise, after removing eligible variables, R is simplified, split into independent clauses if possible, and added to the set of clauses for subsequent resolution cycles. This procedure is continued until a contradiction is detected, thus proving the unsatisfiability of ¬F.

The verification procedure by Bledsoe and Hines is shown to be *complete* [Bledsoe el al 81]. However, completeness requires: 1) mandetory variable elimination, and 2) retention of tautologies. Both requirements tend to degrade the performance of this procedure, and therefore are not employed in practice.

Example 3:

We now show how the safety assertion in Example 2 is shown to be a theorem derivable from the system specification. The approach by Bledsoe and Hines is applied for the mechanical verification of the property. First, the RTL formula consisting of the system specification and the negation of the safety assertion is transformed into clausal form.‡ Then, the conjunction of these RTL clauses is

---

† The chaining procedure for producing new resolvents is essentially a "limited application" of the transitivity axiom. Specifically, $(a<c)\theta$ is inferred from $a<b$ and $b'<c$ if b or b' is an uninterpreted function containing at least one variable, and b and b' are unifiable with the most general unifier θ.

‡ Most books on automated theorem proving and mathematical reasoning contain a discussion on transforming first-order logic formulas into clausal form (see, e.g., [Chang & Lee 73], [Bundy 83], [Wos et al 84]).

shown to be unsatisfiable, thus proving the safety assertion is consistent with the specification.

The clausal form of the specification and the negation of the safety assertion in Example 2 are shown below. U and T are Skolem constants corresponding to variables u and t, respectively.

Specification:

$$f(x) \leq g_1(x) \qquad \text{(c1)}$$
$$g_2(z) \leq f(z) + 30 \qquad \text{(c2)}$$
$$g_1(y) + 20 \leq g_2(y) \qquad \text{(c3)}$$

Negation of Safety Assertion:

$$g_2(U) \leq h(T) \qquad \text{(c4)}$$
$$h(T) \leq g_2(U) + 10 \qquad \text{(c5)}$$
$$h(T) \leq f(U) \vee f(U) + 41 \leq h(T) \qquad \text{(c6)}$$

Unsatisfiability Proof:

1.	$g_2(U) \leq f(U) \vee f(U) + 41 \leq h(T)$	Chaining, c4, c6
2.	$g_2(U) \leq f(U) \vee f(U) + 31 \leq g_2(U)$	Chaining, 1, c5
3.	$g_2(U) \leq g_1(U) \vee f(U) + 31 \leq g_2(U)$	Chaining, 2, c1
4.	$g_2(U) + 20 \leq g_2 \vee f(U) + 31 \leq g_2(U)$	Chaining, 3, c3
5.	$f(U) + 31 \leq g_2(U)$	1st literal in 4 unsat.
6.	$\square$	Chaining, 5, c2

3.3. Remarks on the Undecidability of RTL

RTL in full was shown to be undecidable in[Jahanian et al 88a]. The proof, due to Douglas Stuart, uses a reduction to the acceptance problem for deterministic two counter machines. A formula is constructed which simulates the computation of the machine on a given input, and that is satisfiable if and

only if the machine does not accept the input. Therefore, if a procedure existed to decide the satisfiability of any RTL formula, that procedure could be used to decide the acceptance problem for two-counter machines, which is already known to be undecidable.

The subclass of quantifier-free Presburger arithmetic with uninterpreted integer functions has been shown to be decidable in [Shostak 79]. If a Presburger formula containing uninterpreted functions (or predicates) is universally quantified, Shostak shows that *F* can be reduced to an *equivalid* formula *F'* free of uninterpreted functions. The correctness of the reduction is straightforward; given a model for ¬F, one can construct a model for ¬F', and conversely. Since *F'* is a Presburger formula, its validity can be checked by a decision procedure for Presburger arithmetic.

The class of RTL formulas in [Jahanian & Mok 87] also has a subset for which a decision procedure for determining the satisfiability (or unsatisfiability) of a formula in the subclass can be shown. Suppose an RTL formula is of the form

$$C_1 \wedge C_2 \wedge \cdots \wedge C_n$$

such that each C_i is of the form

$$(Q_i\ L_1 \vee L_2 \vee \cdots \vee L_m)$$

where Q_i is the prefix (quantifiers) for the disjunction and each L_j is an arithmetical relation. Furthermore, suppose that each inequality L_j is of the form

$$\text{occurrence function} \pm \text{integer constant} \leq \text{occurrence function}$$

where the occurrence functions contain integer variables quantified by the corresponding matrix. Note that the negation of an inequality can be replaced by an equivalent inequality without the negation.

Theorem: The above subclass is decidable if the quantifiers in each prefix Q_i

are either all $\forall$ or all $\exists$.

Proof: The proof of this theorem follows directly from Herbrand's Theorem. Suppose *F* is a formula in the above subclass. From the hypothesis, skolemizing *F* produces a set of clauses S such that each skolem function in S is a 0-place function, i.e., a constant. Furthermore, the integer variables in S (denoting occurrences) appear only inside occurrence functions. Since the "@" function does not take an instance of itself as an argument, the Herbrand Universe is finite. By Herbrand's Theorem, the set of clauses in S is unsatisfiable iff there is a finite unsatisfiable set S' of ground instances of S. Since Herbrand's universe is finite, its powerset is finite and so there is a finite number of ground instances of S which must be checked. Checking a finite set of ground clauses can be done using any decision procedure for propositional logic.

□

3.4. Efficiency Considerations

In complex real-time systems, verifying an assertion against the full specification of the system may be inefficient. For some systems, it may be the case that only a subset of the specification can have a bearing on the validity of a particular safety assertion. In this case, much effort can be saved if we can "filter out" the irrelevant assertions from the systems specification before we invoke the primary analysis procedure. For example, if a safety assertion can be expressed in the form of a state predicate about a system attribute S, then we may be able to conclude its validity just from the set of assertions which can directly or indirectly cause a transition of S to occur. If we can determine such a set of supporting assertions by an efficient procedure, e.g., by computing a transitive closure using the accountability and causal assertions, then the primary procedure may only have to contend with a much smaller set of axioms.

Another possible source of efficiency gain is the operating characteristics of the real-time system itself. The operation of many process control systems is often structured according to different *modes*. Intuitively, a mode corresponds to some assertion(s) about the values of a set of system attributes, e.g., an airplane is in emergency landing mode if the hydraulic pressure in the landing gear subsystem is below a critical value or if the fuel gauge indicates an empty tank. The key here is that the assertions in a system specification are often qualified by a collection of modes, i.e., they are required to be true only under those modes. Under other modes, these assertions need not be true. If there is an easy way for us to deduce that a given safety assertion is true in certain modes and these modes include all the ones that qualify a certain assertion p in the system specification, then we ought to be able to prove the validity of the safety assertion without making use of p at all.

It is interesting to point out that techniques to improve the efficiency of the inference mechanism such as the ones mentioned above often enable us to prove the validity of a safety assertion independent of whether a feasible schedule (one that meets all the performance requirements of the system) exists.

Although existing procedures for Presburger Arithmetic with uninterpreted functions is suitable for our purposes, greater efficiency may be achieved by inventing a procedure which is tailored specifically for our domain. For example, consider the observation that the RTL assertions described in this paper do not contain algebraic relations involving functions that take an instance of the same function as an argument. An analysis procedure that takes this fact into consideration may indeed provide a more efficient prover for checking ground inequality unit clauses.

For large systems, general techniques for improving the efficiency of the inference mechanism alone are probably inadequate to render the verification of a complete design practical. In such cases, good software engineering methods must be developed to ease the job of design verification by restricting design

complexity. A good design method should encourage designers to structure their systems so that each safety property can be established from a readily determinable subset of the systems specification.

Chapter 4

A Graph-Theoretic Approach for Timing Analysis

This chapter presents a graph-theoretic algorithm for safety analysis of a class of timing properties in real-time systems which are expressible in a subset of Real Time Logic (RTL) formulas. Our procedure is in three parts: The first part constructs a graph representing the system specification and the negation of the property to be verified. The second part detects positive cycles in the graph using a node removal operation. The third part determines the consistency of the desired property with respect to the system specification based on the positive cycles detected. The implementation and an application of this procedure will also be described.

4.1. Motivation

In the preceding chapter, we presented a formal approach based on RTL for the specification and verification of timing properties in real-time systems. A property under investigation is shown to be consistent with respect to a system specification by proving that the conjunction of the RTL formulas representing the system and the negation of the timing property is unsatisfiable. After transforming the RTL formulas into predicates in Presburger Arithmetic with uninterpreted integer functions, it was shown that well-known procedures (such as the resolution-based system in [Bledsoe & Hines 80]) for proving theorems about general linear inequalities may be used to perform the analysis. These procedures are in general semi-decision procedures since Presburger Arithmetic with even a single uninterpreted function is undecidable [Downey 72]. In practice, however, it has been observed that existing procedures work quite well with at least small problem instances.

Since a decision procedure for Presburger Arithmetic is at least double exponential, any proof method for RTL based on existing methods is likely to be impractical for large problems. In this chapter, we shall present a more practical procedure for the verification of timing properties which are expressible in a subset of RTL. While our algorithm (like other well-known algorithms for Presburger arithmetic with uninterpreted functions) is not a decision procedure, it exploits an efficient constraint-graph technique in integer programming and should work substantially faster for modularly designed systems for which the violation of a safety property can be related to a small subset of the specification.

There are two primary approaches for attaining greater efficiency. For some systems, it may be the case that only a subset of the specification can have a bearing on the validity of a particular property. Hence, much effort can be saved if we can "filter out" the irrelevant assertions from the system specification before invoking the primary decision procedure. The second approach improves the efficiency of the inference mechanism by considering subclasses of RTL formulas which are not too restrictive in describing the system specifications, but allow the use of more efficient analysis procedures. Alternative ways for improving efficiency based on the first approach will be discussed in the subsequent chapters. This chpater describes a graph-theoretic verification procedure for rather expressive subclass of RTL formulas.

In coming up with this procedure, we were motivated by the following observations:

(1) The RTL formulas describing real-time systems in many cases consist of arithmetic inequalities (which may be quantified) involving two terms and an integer constant where a term is either a variable or a function.

(2) The RTL formulas of interest do not involve arithmetic expressions that contain a function taking an instance of itself as an argument.

The preceding observations hold even after RTL formulas are translated into the corresponding formulas in Presburger Arithmetic with uninterpreted functions. The first observation suggests that a graph-theoretic approach may be useful in the analysis. It is well known that an algorithm for finding the shortest paths from a single source, [Lawler 76], can be used for the simple integer programming problem where each inequality is of the form $x_i - x_j \leq \pm a_{ij}$ where x_i and x_j are variables and $\pm a$ is an integer constant. An alternative formulation of this problem allows the construction of a constraint graph for a given set of inequalities. Each variable is represented as a node in the graph, and an inequality $x_i \pm a_{ij} \leq x_j$ is represented by a directed edge with weight $\pm a_{ij}$ connecting x_i to y_j. The given set of inequalities is unsatisfiable if and only if there is a cycle in the graph with positive weight on it. However, the formulas in our analysis involve quantifiers, functions, and disjunctive clauses, thus requiring a more complex representation and a more complicated algorithm.

The organization of this chapter is as follows: The remainder of this section introduces the proposed subclass of RTL formulas. Section 4.2 describes how a special type of graph can be constructed for a given formula consisting of the system specification and the negation of the timing property in question. Section 4.3 gives a few preliminary definitions and describes the relationship between the unsatisfiability of a formula and cycles in the corresponding graph. Section 4.4 presents an algorithm for detecting positive cycles in a graph; Section 4.5 introduces an algorithm for showing the unsatisfiability of the original formula based on the positive cycles detected in the corresponding graph. Section 4.6 discusses the implementation of the analysis procedure and Section 4.7 the mechanical application of the procedure to an example nuclear reactor control rod problem.

4.1.1. Subclass of RTL Formulas

This subsection describes and gives examples of the subclass of RTL formulas for which a verification procedure will be presented in the subsequent sections. Specifically, we restrict ourselves to arithmetic inequalities of the form:

$$\text{occurrence function} \pm \text{integer constant} \leq \text{occurrence function}$$

Of course, variables in an occurrence function can be arbitrarily quantified by $\forall$ and $\exists$. Furthermore, inequalities may be connected using the usual logical connectives ($\wedge$, $\vee$, $\rightarrow$). Negation of an inequality can always be replaced by an equivalent inequality without the negation. For example, $\neg\,(@(E_1,i) \leq @(E_2,j))$ can be replaced by $@(E_2,j) + 1 \leq @(E_1,i)$.

The above subclass allows inequalities involving two occurrence functions and an integer constant. Intuitively, this is consistent with the view that the timing constraints of a real-time system is often represented as imposing an integer constant lower/upper bounds on occurrences of pairs of events, e.g., event E_i occurs 20 time units after occurrence of event E_j, or event E_i occurs at least 10 time units after event E_j.

The RTL formulas of Example 1 in Chapter 3 all subscribe to the above restriction, thus they fall within the subclass. The formula

$$\forall x \exists y \;\; @(\uparrow \text{SAMPLE},x) + y \leq @(\downarrow \text{SAMPLE},x),$$

for example, is not a member of the subclass described above, because the first argument of the predicate $\leq$ is the sum of a function and a variable.

Given a systems specification SP and a safety assertion SA expressed in the subclass of RTL, the objective is to show that SA is consistent with SP by proving that the formula F

$$SP \wedge \neg SA$$

is unsatisfiable. As in Chapter 3, to perform the unsatisfiability proof, we

transform the RTL formula F into the corresponding formula F' in Presburger Arithmetic with uninterpreted integer functions. For each event e, this transformation replaces each occurrence function @(e,i), where i is an integer or a variable, by a function $f_e(i)$. Observe that f_e is an uninterpreted function defined for the event constant e. We then put F' in clausal form and apply our procedure to determine if it is indeed unsatisfiable.

Let F" denote the clausal form of F', i.e., F" is a formula of the form:

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

Each C_i is a disjunctive clause

$$L_1 \vee L_2 \vee \ldots \vee L_m$$

and each L_j is a literal of the form

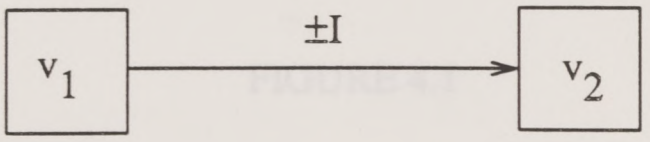
$$v_1 \pm I \leq v_2$$

where v_1 and v_2 are uninterpreted integer functions (replacing the occurrence functions), and I is an integer constant.

4.2. Graph Representation

Given a formula F" in the form described in the previous section, we construct a weighted graph G in which nodes (vertices) denote the terms in F" which are not integer constants, edges represent the literals, and the weights denote the corresponding integer constants in literals.

Consider a literal $v_1 \pm I \leq v_2$. The corresponding graph representation is shown below:



Throughout this chapter, we shall use the letter v to denote vertices in graphs (or corresponding terms in formulas), and $<v_i , v_j>$ to denote a directed edge from node v_i to node v_j.

Nodes representing terms involving the same function symbol will be clustered together so that we can keep track of the consistency among the different instances of the same function. For this purpose, we can view the nodes in a graph G as being partitioned into a collection of disjoint clusters where each cluster is a set containing one or more nodes. All nodes with the same function symbol belong to a single cluster.

Example 1:

Consider the following formula containing two unit clauses:

$f(x) + 10 \leq g(x)$
$g(X) \leq h(X)$

f, g, and h are integer functions symbols,
x is a free variable,
X is a skolem constant.

The corresponding graph is shown in Figure 4.1. Notice that g(x) and g(X) are in one cluster because they share the same function symbol. f(x) and h(X) each belong to a cluster containing only one member. □

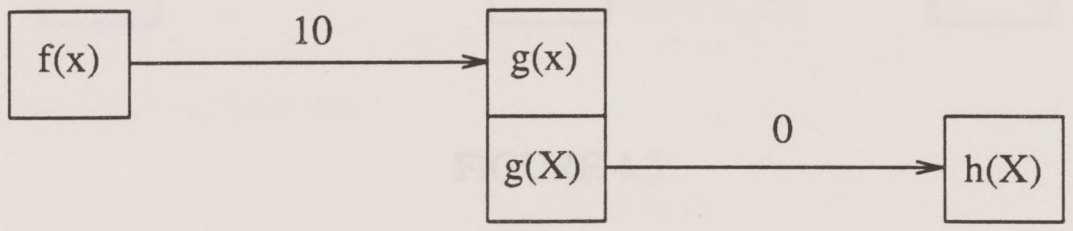


FIGURE 4.1

Given a formula F", Algorithm 1 below describes how the corresponding graph G is constructed. Algorithm 1 iteratively adds all the literals in a clause to the output graph, one clause at a time. For each literal, $v_1 \pm I \leq v_2$, nodes labeled v_1 and v_2 are added to the graph, and a directed edge $<v_1, v_2>$ with weight $\pm I$ is created. Obviously, if either v_1 or v_2 has already been added to the graph, then it is not necessary to have duplicate nodes. For instance, consider adding the literal

$$f(X) + 20 \leq h(X)$$

to the graph in Figure 4.2 which already has a node labeled h(X). Adding the above literal produces the graph in Figure 4.3 with one additional node f(X) and an edge connecting f(X) to h(X) with weight 20.

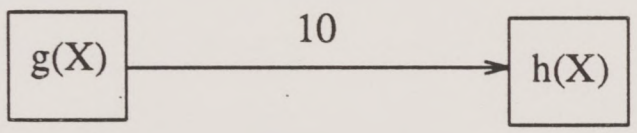


FIGURE 4.2

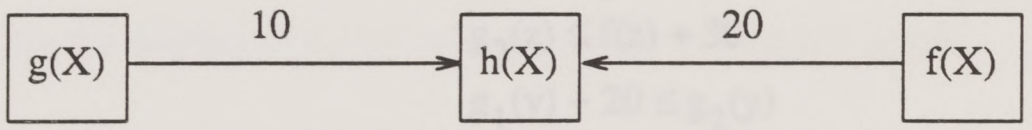


FIGURE 4.3

ALGORITHM 1: Graph Construction

For each clause C_i, For each literal: $v_1 \pm I \leq v_2$,

(1) Find the cluster corresponding to the term v_1.
(The cluster is associated with the function symbol of v_1.)
If the cluster does not exist, create an empty one.

(2) Search the cluster from step 1 for a node labeled v_1.
If not found, add the node to the cluster.

(3) Repeat steps 1 and 2 for the term v_2.

(4) Add a directed edge $<v_1, v_2>$ with weight $\pm I$ from node v_1 to node v_2.

Example 2:

Recall the formula in clausal form presented earlier in Example 3 of Chapter 3:

Specification:

$$f(x) \leq g_1(x)$$
$$g_2(z) \leq f(z) + 30$$
$$g_1(y) + 20 \leq g_2(y)$$

Negation of Safety Assertion:

$$g_2(U) \leq h(T)$$
$$h(T) \leq g_2(U) + 10$$
$$h(T) \leq f(U) \vee f(U) + 41 \leq h(T)$$

Figure 4.4 illustrates the corresponding graph from Algorithm 1. In this example, it is possible to replace y and z by x while constructing the graph, hence producing fewer nodes as shown in Figure 4.4.

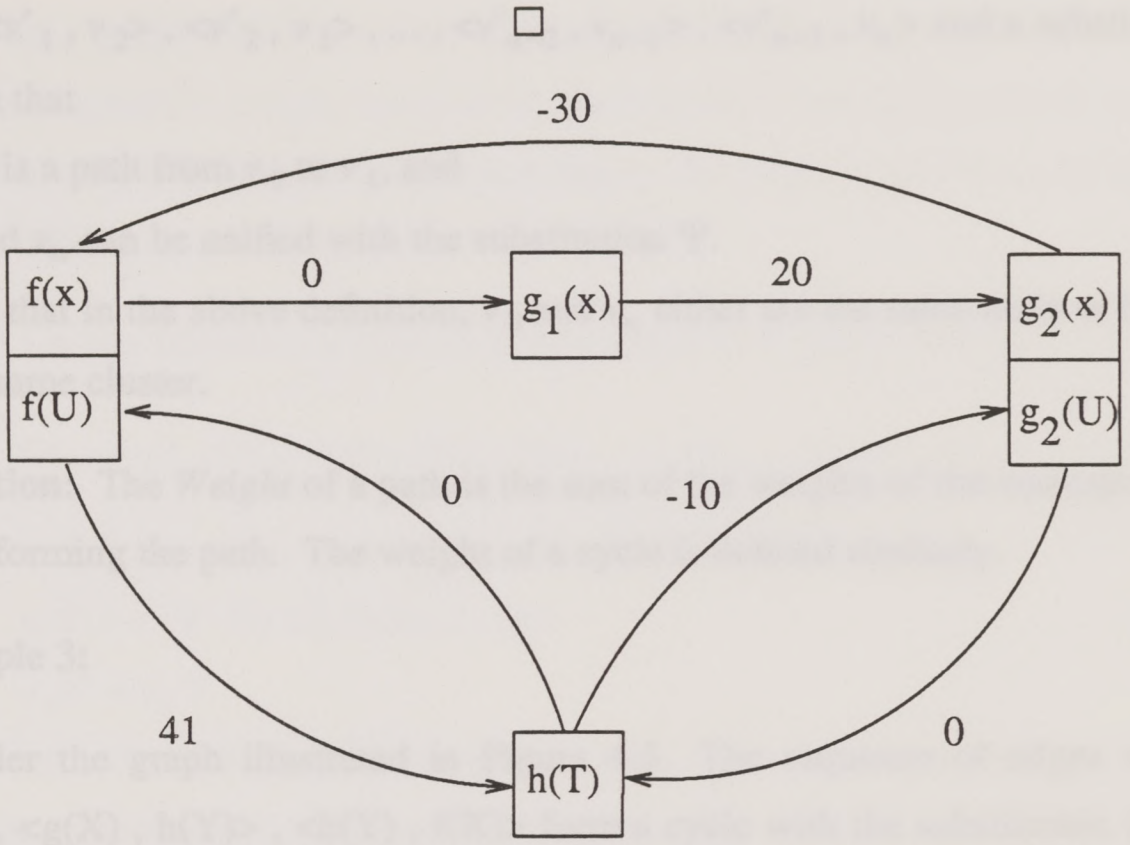


FIGURE 4.4

4.3. Positive Cycles and Unsatisfiability

A graph G, constructed for a given formula F" using algorithm 1, has an important property which can be used to show the unsatisfiability of F". Let us first redefine the notions of a path and a cycle for our special graph.

Definition: Given a pair of nodes v_0 and v_n in a graph G, there is a *path* from v_0 to v_n if there is a sequence of edges $<v_0 , v_1>$, $<v'_1 , v_2>$, $<v'_2 , v_3>$, ... , $<v'_{n-2} , v_{n-1}>$, $<v'_{n-1} , v_n>$ and a substitution Ψ such that pairwise unification of v_i and v'_i for all $1 \leq i < n-1$ can be constructed, i.e., $v_i\Psi = v'_i\Psi$ where $v_i\Psi$ and $v'_i\Psi$ denotes the terms after applying Ψ to v_i and v'_i, respectively.
Observe that the definition requires each pair of v_i and v'_i, $1 \leq i < n-1$, either to be the same or belong to the same cluster.

Definition: There is a *cycle* in a graph G if there is a sequence of edges $<v_0$,

$v_1>$, $<v'_1$, $v_2>$, $<v'_2$, $v_3>$, ... , $<v'_{n-2}$, $v_{n-1}>$, $<v'_{n-1}$, $v_n>$ and a substitution Ψ such that

- there is a path from v_0 to v_1, and

- v_0 and v_n can be unified with the substitution Ψ.

Notice that in the above definition, v_0 and v_n either are the same node or belong to the same cluster.

Definition: The *Weight* of a path is the sum of the weights of the corresponding edges forming the path. The weight of a cycle is defined similarly.

Example 3:

Consider the graph illustrated in Figure 4.5. The sequence of edges <f(x) , g(x)> , <g(X) , h(Y)> , <h(Y) , f(X)> form a cycle with the substitution {X/x}. However, the sequence of edges <f(x) , g(x)> , <g(X) , h(Y)> , <h(x) , f(x)> does not form a cycle, because there is not a substitution which allows the unification of the appropriate terms as required by the definition. □

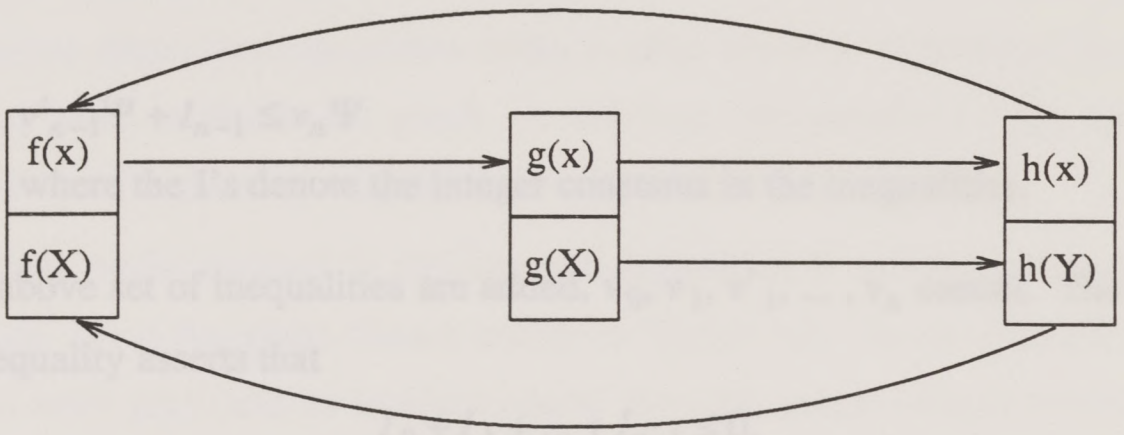


FIGURE 4.5

Theorem 4.1:

Let G be a graph constructed for a given formula F" using Algorithm 1. If a cycle with positive weight exists in G, the formula P consisting of the conjunction of literals (inequalities) corresponding to the edges involved in the cycle is unsatisfiable.

Proof: Let the following sequence of edges denote a positive cycle in a graph:

$$\langle v_0 , v_1 \rangle , \langle v'_1 , v_2 \rangle , \langle v'_2 , v_3 \rangle , \ldots , \langle v'_{n-2} , v_{n-1} \rangle , \langle v'_{n-1} , v_n \rangle$$

From the definition of a cycle, there is a substitution Ψ such that

$v_i\Psi = v'_i\Psi$ for all $1 \le i \le n-1$, and

$v_0\Psi = v_n\Psi$.

Since each edge corresponds to an inequality L_i, one can construct an instance of the conjunction of L_is by applying Ψ to each L_i:

$$\begin{aligned} v_0\Psi + I_0 &\le v_1\Psi \quad \wedge \\ v'_1\Psi + I_1 &\le v_2\Psi \quad \wedge \\ v'_2\Psi + I_2 &\le v_3\Psi \quad \wedge \\ &\vdots \\ v'_{n-1}\Psi + I_{n-1} &\le v_n\Psi \end{aligned}$$

where the I's denote the integer constants in the inequalities.

If the above set of inequalities are added, $v_0, v_1, v'_1, \ldots, v_n$ cancel. The resulting inequality asserts that

$$I_0 + I_1 + \ldots + I_{n-1} \le 0.$$

This is clearly unsatisfiable because the cycle has a positive weight, i.e., sum of the integer constants in the inequalities must be a positive integer. We proved that an instance of the conjunction of the inequalities corresponding to the positive cycle is unsatisfiable, thus we conclude the original set of inequalities is unsatisfiable.

□

Since the literals corresponding to a positive cycle may belong to non-unit disjunctive clauses, the preceding theorem does not guarantee that the detection of a positive cycle in G implies the unsatisfiability of F". As we shall discuss in Section 4.5, it may be necessary to detect several positive cycles to prove the unsatisfiability of a formula F". The next section describes an algorithm for detecting positive cycles in a given graph. The final algorithm for determining the unsatisfiability of F" as positive cycles are detected in the corresponding graph G will follow the next section.

4.4. Detecting Positive Cycles

The algorithm to be described for detecting positive cycles depends on an operation for removing nodes within a cluster such that the positive cycles in the original graph are preserved. Repeating this operation will eventually remove all the nodes and subsequently all the clusters, so that positive cycles can be detected.

Each iteration removes a node and its incident edges, and adds the appropriate edges (and sometimes nodes to other clusters) to preserve the cycles that may have existed in the graph. A self-loop with positive weight denotes a positive cycle in the original graph.

Since node removal is the core of Algorithm 2, we first illustrate this operation using the graph shown earlier in Figure 4.5. As an example, let us remove node g(x) (and its incident edges) from the graph without breaking the existing cycles. Consider the pair of edges <f(x) , g(x)> and <g(x) , h(x)>. Removing g(x) eliminates both edges, thus eliminating the path from f(x) to h(x). Therefore, we need to add an edge from f(x) to h(x). Next consider the pair of edges <f(x) , g(x)> and <g(X) , h(Y)>. Removing g(x) disconnects the path from an instance of f(x), namely f(X), to h(Y) because the edge <f(x) , g(x)> would be removed. Hence, an edge from f(X) to h(Y) is necessary to

preserve the path and the potential cycles. Figure 4.6 illustrates the resulting graph after removing g(x) and its incident edges.

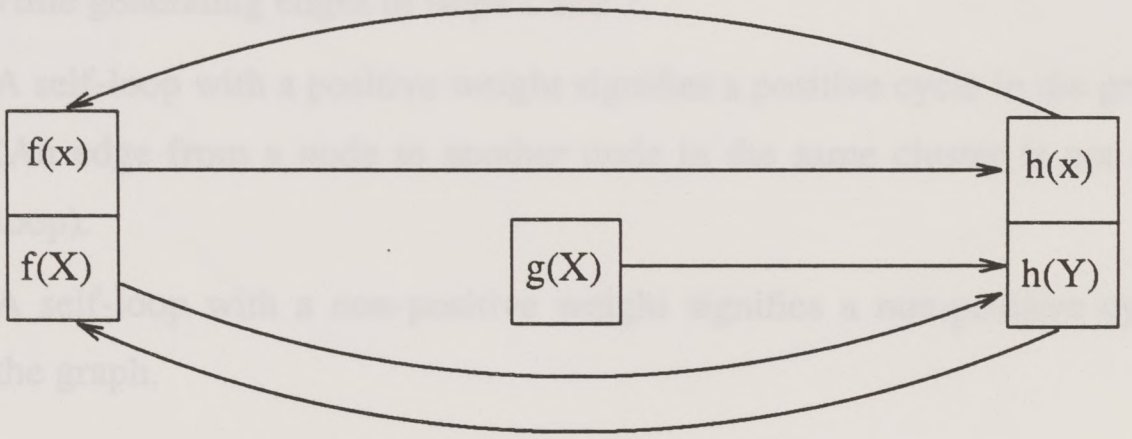


FIGURE 4.6

ALGORITHM 2: Detecting Positive Cycles

For each cluster in the graph,

For each node v in the cluster,

(1) Let $S = \{v_1, v_2, \ldots, v_n\}$ be the cluster to which v belongs, i.e., all the v_is have the same function symbol. Note that S also includes v itself.

For each v_i in S, if v and v_i can be unified,

(a) Let Ψ be the most general unifier of v and v_i. (b) For each pair of non-self-loop edges $\langle v', v\rangle$ and $\langle v_i, v''\rangle$ with weights I_1 and I_2,

- add the nodes $v'\Psi$ and $v''\Psi$ to the respective clusters if not already there and add the edge $\langle v'\Psi, v''\Psi\rangle$ with weight I_1+I_2, where

$v'\Psi$ denote the label for a node after applying Ψ to v',

$v''\Psi$ denote the label for a node after applying Ψ to v''.

(c) Repeat step (b) for each pair of non-self-loop edges $<v' , v_i>$ and $<v , v''>$.

(2) Remove node v and all edges incident on it from the graph.

(3) While generating edges in steps 1 and 2,

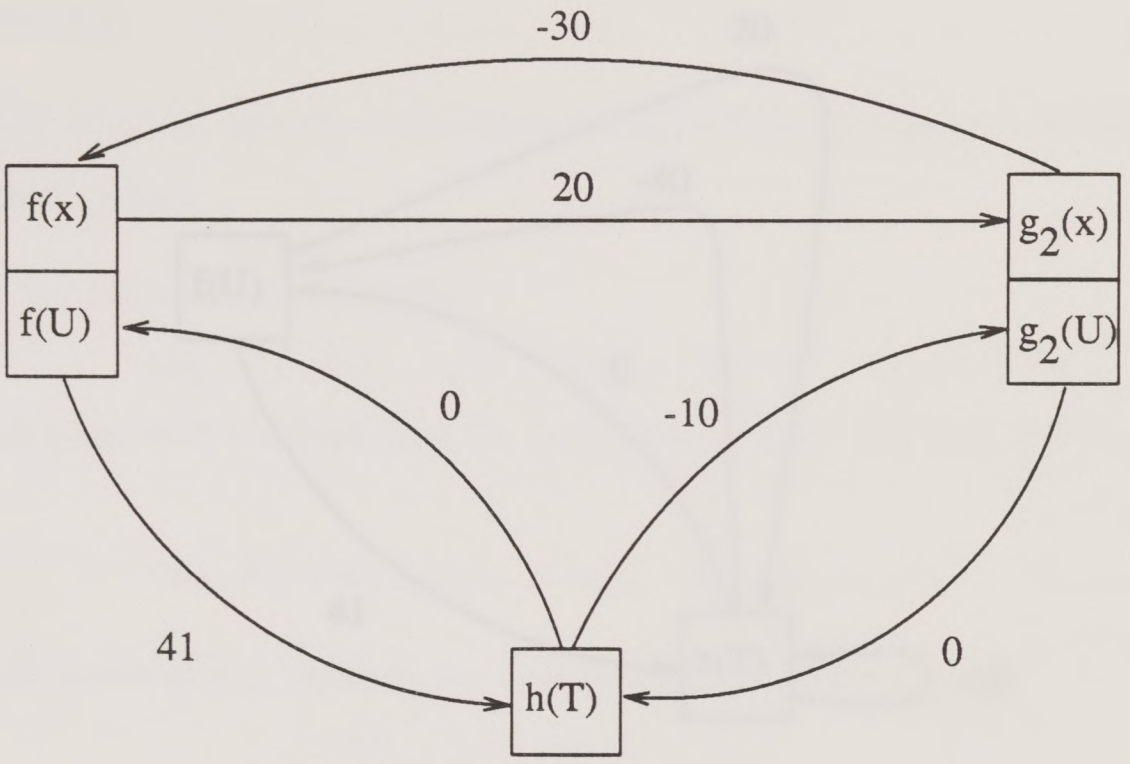
A self-loop with a positive weight signifies a positive cycle in the graph. (An edge from a node to another node in the same cluster is not a self-loop).

A self-loop with a non-positive weight signifies a non-positive cycle in the graph.

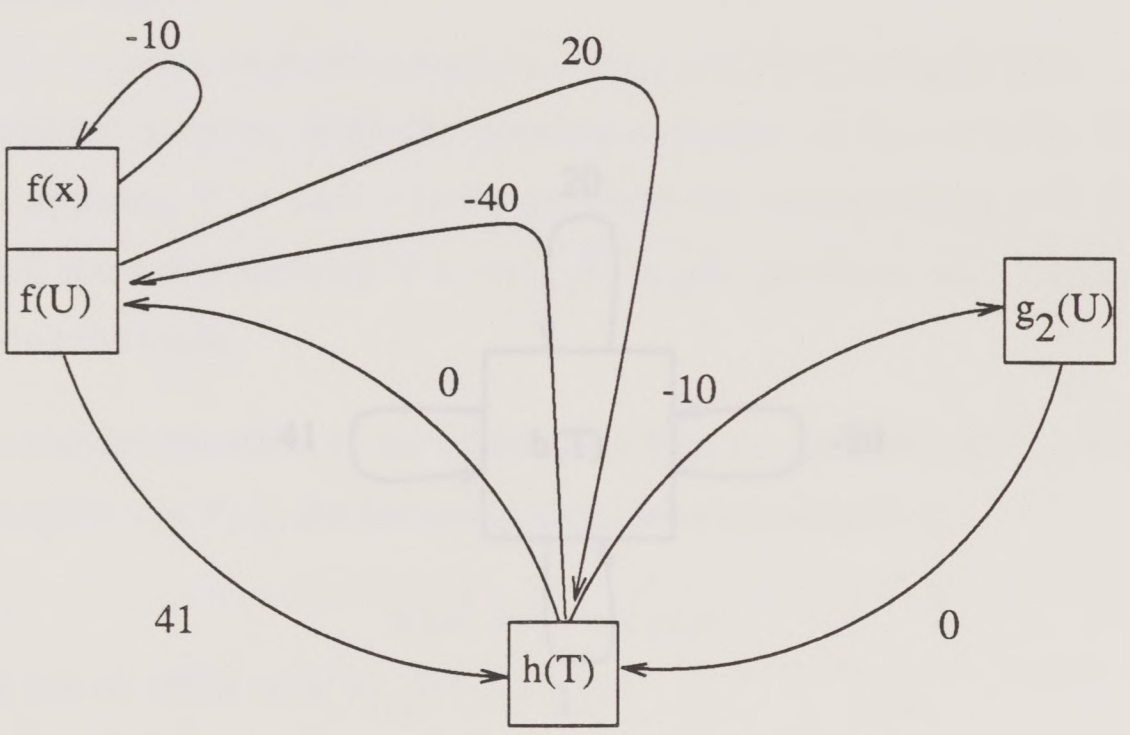
In the the implementation of this algorithm, after an edge is added to the graph, a note is made of the two edges combined to generate the new edge. This information is important in identifying the edges from the initial graph involved in a cycle after detection. The reader may question why the negative cycles are not discarded in step 3. As it will be described in Section 4.5, certain non-positive cycles are used in the unsatisfiability checker.

Example 4:

Let us apply Algorithm 2 to detect the positive cycles in the graph of Figure 4.4. Figures 4.7(a-d) illustrate the resulting graphs after the repeated applications of node removal operation. Observe that non-positive cycles are ignored in this example. As shown in Figure 4.7(d), three positive cycles were detected. □

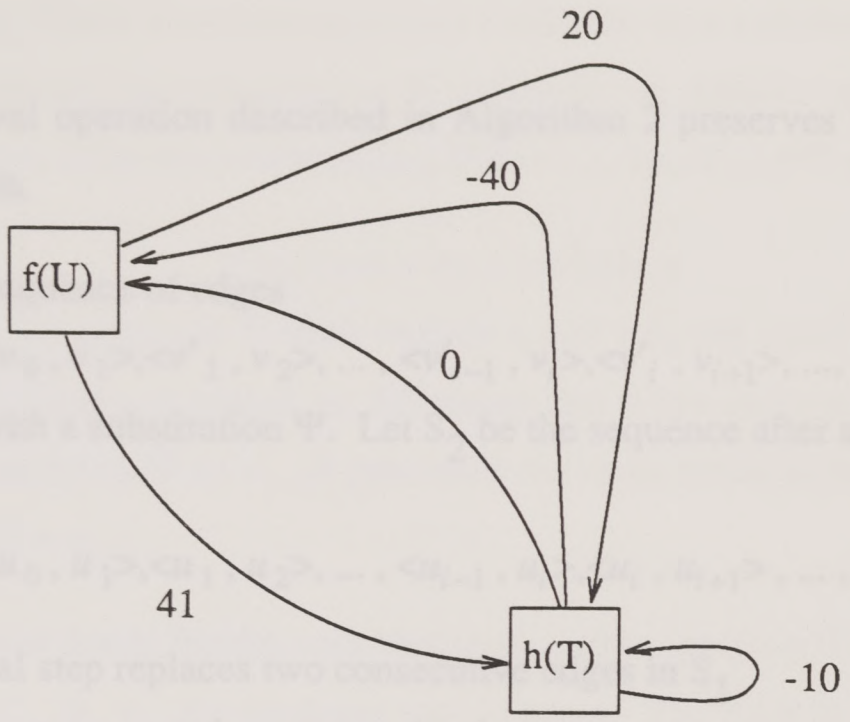


(a) After removing node $g_1(x)$.

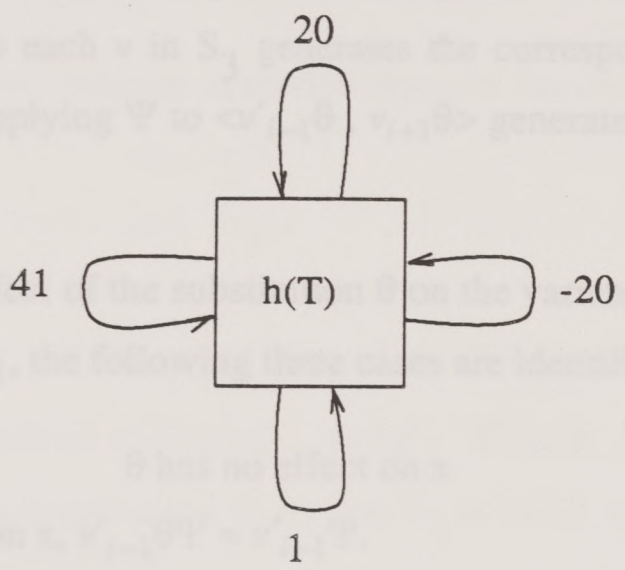


(b) After removing node $g_2(x)$.

FIGURE 4.7



(c) After removing nodes $g_2(U)$ and f(x).



(d) After removing node f(U).

FIGURE 4.7

Theorem 4.2:

The node removal operation described in Algorithm 2 preserves the cycles of the original graph.

Proof: Let the sequence of edges

S_1: $\langle v_0 , v_1\rangle,\langle v'_1 , v_2\rangle, \ldots , \langle v'_{i-1} , v_i\rangle,\langle v'_i , v_{i+1}\rangle, \ldots, \langle v'_{n-1} , v_n\rangle$

denote a cycle with a substitution Ψ. Let S_2 be the sequence after applying Ψ to S_1:

S_2: $\langle u_0 , u_1\rangle,\langle u_1 , u_2\rangle, \ldots , \langle u_{i-1} , u_i\rangle,\langle u_i , u_{i+1}\rangle , \ldots , \langle u_{n-1} , u_n\rangle$

The node removal step replaces two consecutive edges in S_1

$$\langle v'_{i-1} , v_i\rangle \text{ and } \langle v'_i , v_{i+1}\rangle$$

by a single edge $\langle v'_{i-1}\theta , v_{i+1}\theta\rangle$ where θ is the most general unifier of v_i and v'_i. We prove the theorem by showing that applying the substitution Ψ to the sequence of edges

S_3: $\langle v_0 , v_1\rangle,\langle v'_1 , v_2\rangle, \ldots , \langle v'_{i-1} , v_{i+1}\rangle, \ldots , \langle v'_{n-1} , v_n\rangle$

still produces a cycle. With the possible exception of the terms in $\langle v'_{i-1}\theta , v_{i+1}\theta\rangle$, applying Ψ to each v in S_3 generates the corresponding u in S_2. We must still show that applying Ψ to $\langle v'_{i-1}\theta , v_{i+1}\theta\rangle$ generates $\langle u_{i-1} , u_{i+1}\rangle$, thus preserving the cycle.

Let us consider the effect of the substitution θ on the variables of v'_{i-1} first. For each variable x in v'_{i-1}, the following three cases are identified:

Case 1: θ has no effect on x

Since θ has no effect on x, $v'_{i-1}\theta\Psi = v'_{i-1}\Psi$.

Hence, $v'_{i-1}\theta\Psi = u_{i-1}$.

Case 2: θ replaces x by a non-variable term t

Note that t is either a constant or a function. Since θ is a most general unifier, the substitution Ψ must also replace x by t (or an instance of t if t is a non-

ground function). Since the same argument holds for each variable in t, applying θ does not violate the substitution Ψ makes for x. Hence, applying Ψ to $v'_{i-1}\theta$ generates the same term as applying Ψ to v'_{i-1}, i.e., $v'_{i-1}\theta\Psi = u_{i-1}$.

Case 3: θ replaces x by a variable y

Since $v_i\theta = v'_i\theta$, we conclude Ψ must replace x and y by the same term (or one by the other). Therefore, it makes no difference that θ first replaces x by a variable y. Hence, applying Ψ to $v'_{i-1}\theta$ generates the same terms as applying Ψ to v'_{i-1}, i.e., $v'_{i-1}\theta\Psi = u_{i-1}$.

The same argument holds for the variables in v_{i+1} after applying θ to it. Hence, applying Ψ to $<v'_{i-1}\theta , v_{i+1}\theta>$ produces $<u_{i-1} , u_{i+1}>$. The cycle is preserved.

□

Theorem 4.3:

The node removal operation described in Algorithm 2 does not introduce any new cycle which does not exist in the original graph.

Proof: Suppose the node removal operation replaces the two edges e_1 and e_2

e_1: $<v_i , v_j>$

e_2: $<v'_j , v_k>$

by an edge e_3

e_3: $<v_i\theta , v_k\theta>$

where θ is the most general unifier of v_j and v'_j. If e_3 is involved in a positive cycle, one can apply a substitution to e_1 and e_2 to make the two edges participate in the same cycle.

□

We now discuss the termination condition for Algorithm 2. Recall that two types of functions appear in the clausal form of our formulas: occurrence

functions corresponding to particular events, and skolem functions appearing as arguments to the occurrence functions. Algorithm 2 as stated above may not terminate if a function is allowed to take itself as an argument. We add the condition to ensure that a function symbol does not appear more than once in a term corresponding to a node in our graph. Step 1 of Algorithm 2 is rewritten to include this condition:

216.nr 99 12 (1) Let S = $\{v_1, v_2, \ldots, v_n\}$ be the cluster to which v belongs.

For each v_i in S, if v and v_i can be unified,

(a) Let Ψ be the most general unifier of v and v_i.

(b) For each pair of non-self-loop edges $\langle v', v\rangle$ and $\langle v_i, v''\rangle$ with weights I_1 and I_2, if a function symbol does not appear more than once in $v'\Psi$ and $v''\Psi$,
add the nodes $v'\Psi$ and $v''\Psi$ to the respective clusters if not already there and add the edge $\langle v'\Psi, v''\Psi\rangle$ with weight I_1+I_2.

(c) Repeat step (b) for each pair of non-self-loop edges $\langle v', v_i\rangle$ and $\langle v, v''\rangle$.

The added condition in step 1(b) that a function symbol may not appear more that once in $v'\Psi$ and $v''\Psi$ ensures the termination of Algorithm 2. (Same condition is also imposed on step 1(c)). Since an occurrence function does not take an instance of itself as an argument, the condition in step 1(b) does not impose any restriction on the occurrence functions. However, the condition prevents a skolem function from taking an instance of itself as an argument. For example, a node labelled f(h(h(x))) is not allowed in our constraint graph where f denotes an occurrence function, and h is a skolem function.

Since a skolem function replaces an existentially quantified variable, the clausal form of the initial set of formulas under investigation does not have any

term in which a skolem function appears more than once. Hence, the condition in step 1(b) does not restrict the subclass of formulas described in this paper. However, the node removal operation may in some cases produce nodes in which a skolem function symbol appears more than once. The condition in step 1(b) prevents the generation of such terms. Although completeness cannot be claimed for Algorithm 2 under this condition, we impose this condition in the implementation of this algorithm since it seems to be a good compromise in practice.

It is straightforward to show that Algorithm 2 terminates. In each iteration, nodes in a cluster are removed one by one until the cluster is empty. However, as nodes in a cluster are removed, new nodes may be added either to the same cluster or to the remaining clusters (not yet processed). The number of nodes which are added to the same cluster cannot grow without a bound because a skolem function symbol is not allowed to appear more than once in any node. A node which is added to a remaining cluster is eventually removed when the respective cluster is being processed. Furthermore, after all the nodes in a cluster are removed, the corresponding (occurrence) function symbol does not appear in any node. Hence, all clusters are eventually removed, thus ensuring termination.

4.5. Unsatisfiability Proof

Thus far we have seen how to construct a graph G from a formula F" and to detect positive cycles in the graph. This section presents an algorithm for determining unsatisfiability of F" as positive cycles are detected in the corresponding graph G. It is straightforward to show that if all edges involved in a positive cycle in G correspond to literals (inequalities) which belong to unit clauses, F" must be unsatisfiable. The case where one edge corresponds to a literal (inequality) which belongs to a non-unit disjunctive clause C_i is more difficult; it must be shown that each of the remaining literals in C_i is also

involved in a different positive cycle.

For instance, recall the last clause in Example 2:

$$h(T) \leq f(U) \vee f(U) + 50 \leq h(T)$$

If the edge corresponding to the literal $h(T) \leq f(U)$ is involved in a positive cycle, it must be shown that the edge for $f(U) + 50 \leq h(T)$ is also involved in a positive cycle.

The problem becomes increasingly computationally intensive when more edges in a positive cycle belong to non-unit disjunctive clauses. It will be seen shortly that this problem is as hard as the CNF satisfiability problem of propositional logic where each disjunctive clause contains either all negated literals or all unnegated (positive) literals.

The set of all clauses in a formula F" can be viewed as a collection (conjunction) of m clauses $C_1, C_2, \ldots, C_m$ where each C_k is a disjunctive clause

$$L_{k,1} \vee L_{k,2} \vee \ldots \vee L_{k,n_k}$$

Note that this is merely a notational convention and we are not requiring literals in different clauses be distinct.

Let the following notation denote the list of literals, i.e., inequalities, corresponding to the edges in the i^{th} positive cycle detected.

$$X_{i,1}, X_{i,2}, \ldots, X_{i,m_i}$$

where $X_{i,j}$ signify the j^{th} literal (edge) in the i^{th} positive cycle detected and each $X_{i,j}$ is a literal in at least one of the C_ks. Let P_i denote the formula consisting of the conjunction of literals $X_{i,j}$s:

$$X_{i,1} \wedge X_{i,2} \wedge \ldots \wedge X_{i,m_i}$$

From Theorem 4.1 we conclude that P_i is unsatisfiable. Consequently, F" is satisfiable if and only if $F" \wedge \neg P_i$ is satisfiable. Hence, the positive cycle can be

viewed as adding the clause $\neg P_i$, i.e.,

$$\neg X_{i,1} \vee \neg X_{i,2} \vee \ldots \vee \neg X_{i,m_i}$$

to the existing set of clauses. Now we can use $\neg P_i$ to refute F".

Theorem 4.4: *(Variation of Herbrand's Theorem)*

A set S of clauses is unsatisfiable if and only if there is a finite unsatisfiable set of ground instances of S and $\neg P_i$ for $1 \leq i \leq n$ where each P_i is the conjunction of inequalities corresponding to the edges in a positive cycle detected in the constraint graph for S.

The above formulation permits one to use any method in propositional logic to check for unsatisfiability as positive cycles are detected and the appropriate clauses are added to the existing set of clauses. For example, the Davis-Putnam method is one of the more efficient techniques for testing the unsatisfiability of CNF formulas in propositional calculus, see Chapter 1 in [Sahni 85] or pp. 62-66 in [Chang & Lee 73]. Since it is well known that CNF satisfiability for propositional logic is NP-complete, one may hope that the special form of our clauses (each clause in our the problem contains either only negated literals or only un-negated literals: the literals $X_{i,j}$ in clauses denoting positive cycles are all negated, and the literals in each C_k are all unnegated) might lend itself to a more efficient decision procedure. Unfortunately, we can show:

Theorem 4.5:

The CNF satisfiability of the propositional logic is NP-complete even if each clause contains either all negated or all unnegated (positive) literals.

Proof: It is known that the CNF satisfiability problem for the propositional logic where a clause may contain both negated and un-negated literals is NP-

complete. A simple transformation maps any instance of this problem into an equivalent instance of the CNF satisfiability problem in which each clause contains only negated or only un-negated literals. The idea is to replace a negated literal $\neg L$ by an un-negated literal L' and add the following clauses:

$$L \vee L'$$
$$\neg L \vee \neg L'$$

Clearly, the transformation can be done in polynomial time. □

The remainder of this section presents an algorithm for testing the unsatisfiability for formulas in which each clause contains either only negated or only unnegated literals. Our algorithm is more suited for safety analysis of real-time systems for the following reasons.

First, it is desirable to have a dynamic algorithm in the sense that as each new cycle is detected, the corresponding clause is added to the set of existing ones, and is then checked for unsatisfiability. If it is shown to be unsatisfiable, we can stop at once. Otherwise, we need to continue the node removal operation until another positive cycle is found. The well-known procedures (including the Davis-Putnam method) require redoing the computation each time a new clause is added, i.e., a positive cycle is detected. In contrast, the algorithm to be described builds on top of the computation already done to check for unsatisfiability as each new positive cycle is detected.

Second, even though the algorithm to be described has an exponential time complexity in the *worst* case as one may expect, it is desirable to have an algorithm whose running time in the *worst* case is exponential with respect to the number of positive cycles detected rather than being exponential with respect to the number of literals or the total number of clauses, the rationale being that we expect to have only a few positive cycles detected in most cases.

Finally, while checking for satisfiability after detecting a positive cycle, if the formula is still satisfiable, the proposed algorithm may terminate very quickly after generating only a small part of the search tree. In contrast, the Davis-Putnam method must still complete removing almost all of the literals before concluding satisfiability.

Before presenting a detailed description, it helps to illustrate the algorithm by means of an example. Consider the following set S_1 of clauses with only unnegated literals (this corresponds to the set of original clauses C_is),

C_1: A ∨ B ∨ C
C_2: A ∨ F
C_3: D ∨ C
C_4: B ∨ E
C_5: F ∨ C
C_6: F ∨ E

and the following set S_2 of clauses with only negated literals (each clause corresponds to a positive cycle detected),

¬A ∨ ¬D ∨ ¬F
¬B ∨ ¬F
¬C ∨ ¬E

The goal of the algorithm is to construct a search tree from the set S_2 and determine the unsatisfiability of the collection of clauses in S_1 and S_2 while building the tree. Figure 4.8 illustrates a *worst case* search tree constructed for the set S_2. Each node in the tree corresponds to the collection of one or more literals in S_2. The nodes in the first level correspond to the literals in the first clause of S_2, i.e., literals ¬A, ¬D, and ¬F. To construct the second level, we added the literals in the second clause of S_2, i.e., literals ¬B and ¬F, to each subtree rooted by ¬A, ¬D, and ¬F. The next level was constructed similarly. Thus, each leaf node in a *worst case* search tree corresponds to a conjunction in the disjunctive normal form of the clauses in S_2.

To show that the collection of clauses in S_1 and S_2 is unsatisfiable, one must show that each leaf node in the tree meets one of the following two conditions:

(1) the conjunction of literals (inequalities) in the leaf node makes at least one clause in S_1 unsatisfiable, or

(2) the conjunction of literals (inequalities) in the leaf node is unsatisfiable by itself.

Let us first discuss condition 1. Observe that any node in the search tree makes the set of clauses in S_1 unsatisfiable iff there exists a clause C_i in S_1 such that the negated form of each literal in C_i is also a literal in the label of that node. For instance, the leftmost leaf node, labeled ($\neg A$, $\neg B$, $\neg C$), makes S_1 unsatisfiable because the negated form of every literal in clause C_1 is also represented in the leaf node. Since every leaf node in the tree in Figure 4.8 makes S_1 unsatisfiable, we conclude that the conjunction of clauses in S_1 and S_2 is unsatisfiable.

Fortunately, in most cases we do not have to generate the entire search tree to conclude unsatisfiability by condition 1. In particular, we make use of two important properties to trim the search tree. First, if a node makes S_1 unsatisfiable, then every node in the subtree rooted by that node has the same property, thus it is not necessary to generate the rest of the nodes in the subtree. For instance, the nodes in the subtree rooted by the node labeled ($\neg A$, $\neg F$) need not be generated because ($\neg A$, $\neg F$) makes S_1 unsatisfiable. This property clearly reduces the size of the search tree in many cases.

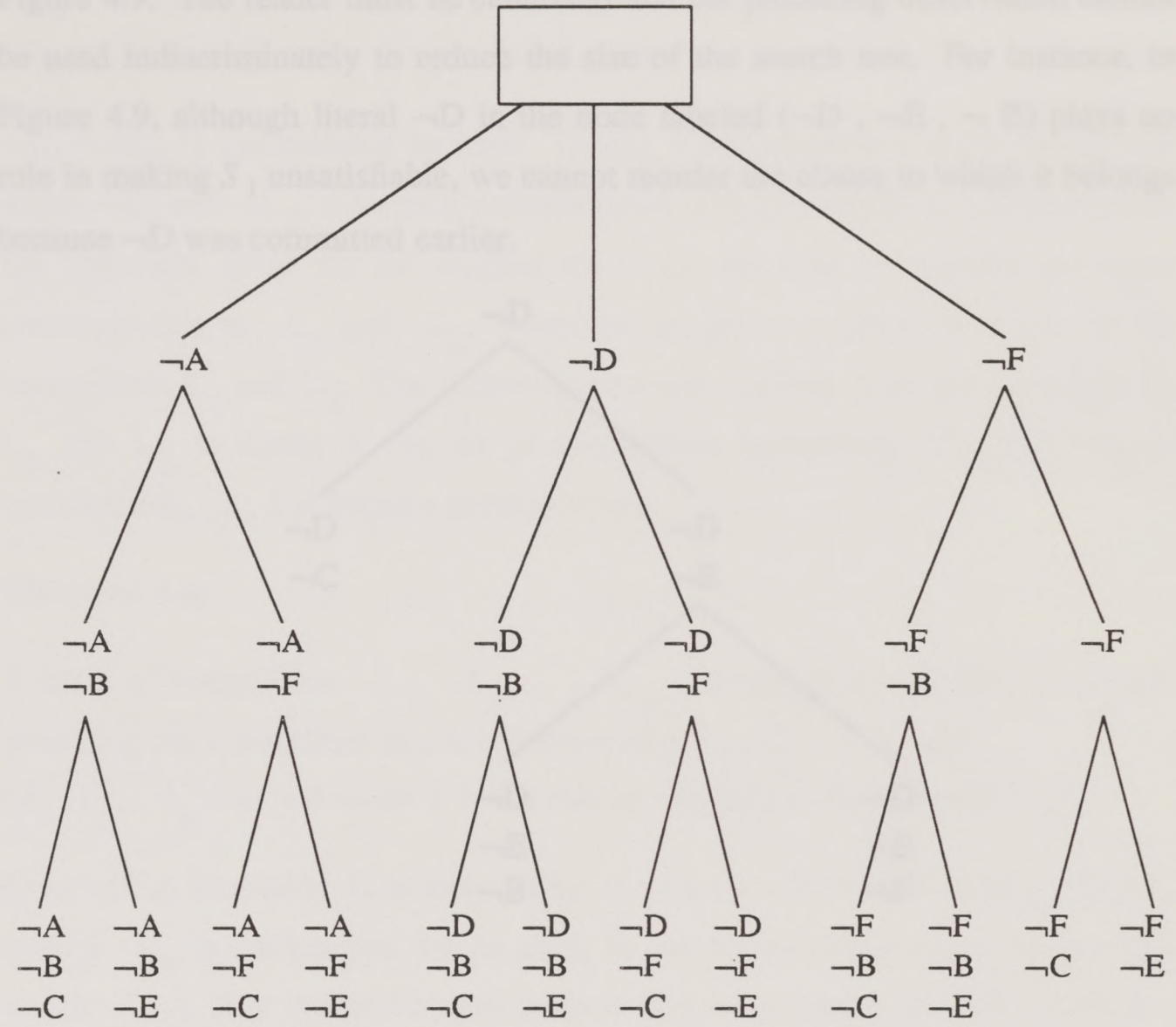


FIGURE 4.8

Another equally important property of the search tree further helps us in reducing the size. Consider the subtree rooted by the node labeled ¬D in Figure 4.8. The leftmost node in that subtree, labeled (¬D , ¬B , ¬C), makes S_1 unsatisfiable because the negated form of every literal in clause C_3 of S_1 is also present in the leftmost node. However, observe that literal ¬B does not have any effect here. Therefore, we can consider the positive cycle denoted by the clause ¬C ∨ ¬E first, and generate a smaller subtree rooted by ¬D as shown in

Figure 4.9. The reader must be cautioned that the preceding observation cannot be used indiscriminately to reduce the size of the search tree. For instance, in Figure 4.9, although literal ¬D in the node labeled (¬D , ¬E , ¬ B) plays no role in making S_1 unsatisfiable, we cannot reorder the clause to which it belongs because ¬D was committed earlier.

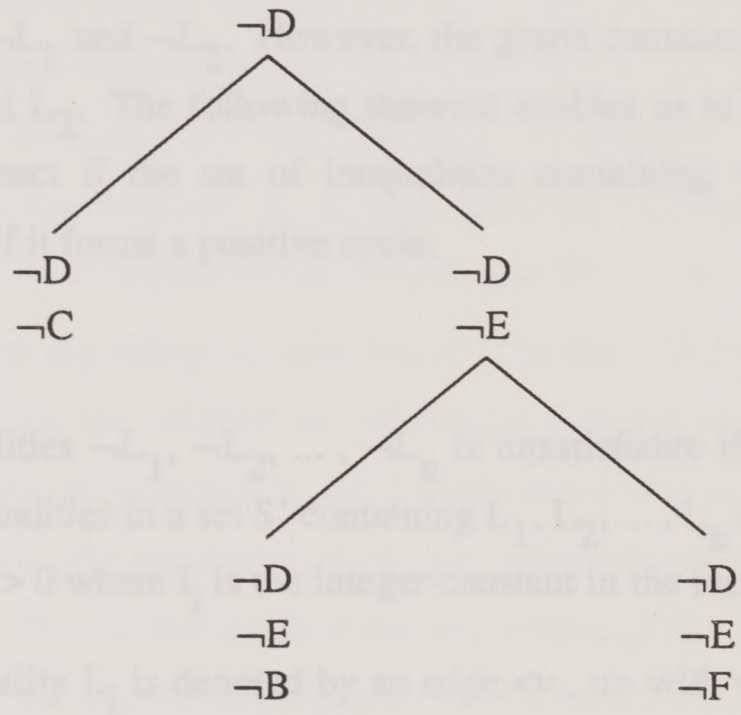


FIGURE 4.9

Let us now discuss condition 2. Since each literal in a leaf node is a negation of an inequality, the conjunction of these literals may be unsatisfiable by itself. For instance, a leaf node consisting of the two literals

$\neg L_1$: $\neg (f(a) \leq g(a))$

$\neg L_2$: $\neg (g(a) \leq f(a))$

may not make any of the clauses in the original set of formulas unsatisfiable. However, if we rewrite the two literals as

$\neg L_1$: $g(a) + 1 \leq f(a)$

$\neg L_2$: $f(a) + 1 \leq g(a)$,

$\neg L_1 \wedge \neg L_2$ is unsatisfiable by itself. In fact, if we construct a graph for the two edges $\neg L_1$ and $\neg L_2$, it forms a positive cycle with the weight 2. Unfortunately, the constraint graph for the original set of clauses may not contain the edges corresponding to $\neg L_1$ and $\neg L_2$. However, the graph contains the edges for the inequalities L_1 and L_2. The following theorem enables us to use the edges for L_1 and L_2 to detect if the set of inequalities containing $\neg L_1$ and $\neg L_2$ is unsatisfiable, i.e., if it forms a positive cycle.

Theorem 4.6:

A set S of inequalities $\neg L_1, \neg L_2, \ldots, \neg L_n$ is unsatisfiable iff there is a cycle involving the inequalities in a set S' containing $L_1, L_2, \ldots, L_n$ and $(-I_1 - I_2 \ldots - I_n + n) > 0$ where I_i is the integer constant in the inequality L_i.

Proof: If an inequality L_i is denoted by an edge <v , u> with weight I_i, the inequality $\neg L_i$ is represented by an edge in the opposite direction, <u,v>, with weight $-I_i+1$. It is straightforward to show that $\neg L_i$s form a cycle iff L_is form a cycle. The sum of the weights for the backward edges, i.e. weights for $\neg L_i$s, must be positive to obtain unsatisfiablity. Hence, $(-I_1 - I_2 \ldots - I_n + n) > 0$.□

Due to the above theorem, we can modify Algorithm 2 to detect if there are cycles involving $\neg L_i$s. The following step can be added to Algorithm 2:

(4) After detecting a cycle with weight *W* consisting of n edges, check if $-W+n > 0$. If so, conclude that the negation of the literals in the cycle form a positive cycle.

Let us now show how to check for condition 2 using the positive cycles detected in step 4. Each new positive cycle in step 4 implies that a set of inequalities

$$\neg L_1, \neg L_2, \ldots, \neg L_n$$

is unsatisfiable. (Note that each L_i is an inequality from the original set of clauses.) One can view this new positive cycle as

$$\neg (\neg L_1 \wedge \neg L_2 \wedge ... \wedge \neg L_n)$$

or alternatively as the following clause

$$L_1 \vee L_2 \vee ... \vee L_n$$

Observe that the above clause contains literals which are all unnegated. Hence, if we add the clauses obtained in step (4) of Algorithm 2 to the original set of clauses just prior to testing for unsatisfiability (Algorithm 3), it will *not* be necessary to check for condition 2. In fact, checking the leaf nodes for condition 1 is sufficient while generating the tree as described above.

A few remarks regarding the termination condition for Algorithm 3 are in order. Let S_1 denote the original set of clauses and the clauses which were added due to step (4) of Algorihm 2. Let S_2 denote the clauses corresponding to the positive cycles in the constraint graph, i.e., positive cycles detected in step (3) of Algorihm 2. If each leaf node in the tree generated for S_2 makes a clause in S_1 unsatisfiable, the algorithm terminates to conclude $S_1 \wedge S_2$ is unsatisfiable. On the other hand, while generating the search tree, if we reach a leaf node that does not make S_1 unsatisfiable and there is no more clause in S_2 to be considered, then satisfiability is possible. The implication is that additional clauses, i.e. positive cycles, are needed to prove unsatisfiability.

Algorithm 3 is the depth-first search approach to the construction of the search tree. The two properties mentioned earlier are embbeded in the algorithm to trim the tree.

ALGORITHM 3: Testing Unsatisfiability

Let $X_{i,j}$ denote the j^{th} literal in the i^{th} clause (positive cycles). Let m_i denote the number of literals in the i^{th} clause (positive cycle).

Let α denote the list of all the literals committed at a given point in the search

tree, i.e., α represents the current leaf node in the search tree for which unsatisfiability is yet to be shown.

$$\alpha = (X_{i_1,j_1}, X_{i_2,j_2}, \ldots, X_{i_p,j_p})$$

Let β denote the list of the first literals of all clauses (positive cycles) which are being considered in the search tree.

$$\beta = (X_{a_1,1}, X_{a_2,1}, \ldots, X_{a_q,1})$$

Let γ denote the list of the first literals of all clauses (positive cycles) which either were rejected or yet to be considered.

$$\gamma = (X_{b_1,1}, X_{b_2,1}, \ldots, X_{b_r,1})$$

Upon detection of a positive cycle, Algorithm 3 is used to test for unsatisfiability. Initially, upon detection of the first positive cycle, the main routine below is invoked with the following parameters:

α = empty
β = empty
$\gamma = (X_{1,1})$

Upon detection of the i^{th} positive cycle, the main routine is invoked with following parameters:

α = as left in the previous invocation
β = as left in the previous invocation
γ = add $X_{i,1}$ to γ left in the previous invocation

```
{*** main routine ***}

invoke check(α, β, γ)
if check returns true, then
    exit denoting satifiability   {still satifiable, must wait for the
    next clause with negated literal, i.e., next positive cycle}

{α was shown to be unsatisfiable, now show unsatisfiability
 for the next branch.  Backtrack if necessary}

while α is non-empty,
    get (X_{i,j})   {remove the last element (literal) added to α}
                    {i is set to the clause (positive cycle) number}
                    {j is set to the literal number}

    if j ≠ m_i
        then do
            add X_{i,j+1}  to α
            invoke ckeck(α, β, γ)
            if check returns true, then
                exit denoting satifiability   {still satifiable,
                  must wait for the next positive cycle}
        end
        else  add X_{i,1} to γ
end while

{every leaf node was shown to make S_1 unsatisfiable}
exit denoting unsatisfiability

{*** check function ***}

check (α, β, γ) returns true to denote satisfiability, or
                        false to denote unsatisfiability

for k=1 to r   {note r is number of elements in γ}
    add X_{b_k,1} to β
```

```
   if there is a clause C in the set of clauses S1 such that the
     negated form of every literal in C is either in α or β,
     then do  {a node in the search tree is shown unsatisfiable}
        for each X(i,1) in β that is in C,
           add X(i,1) to α        {the i^th clause is committed}
           remove X(i,1) from γ
        end for
        β = empty
        return(false)   {node denoted by α makes S1 unsatisfiable}
     end
end for

{still satisfiable}
return(true)
end check
```

Example 5:

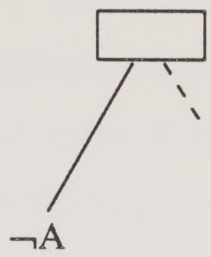
Consider the following set S_1 containing clauses with only unnegated literals,

C_1: A ∨ D
C_2: C ∨ B ∨ D
C_3: A ∨ F
C_4: B ∨ D ∨ E
C_5: B ∨ F
C_6: A ∨ B

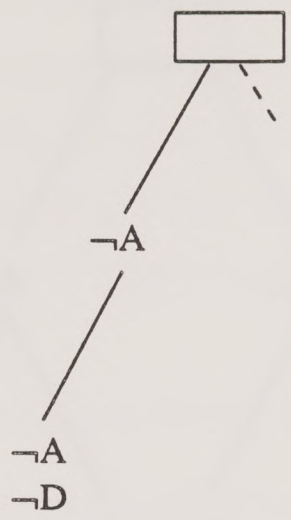
Let ¬A ∨ ¬B be the first clause with only negated literals representing the first positive cycle detected. Figure 4.10(a) illustrates the search tree generated after detecting the first positive cycle. After generating the node labeled ¬A, the construction of the search tree stops because ¬A does not make S_1 unsatisfiable and there is no more positive cycles to consider. Consider three additional clauses with only negated literals, representing the next three positive cycles detected.

¬D ∨ ¬F
¬A ∨ ¬C
¬B ∨ ¬E

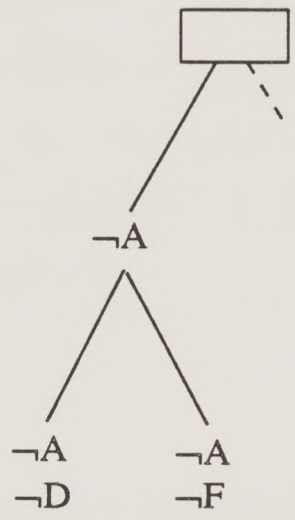
Figure 4.10(b) shows the search tree after generating the leaf node at the next level. Observe that the leaf node ($\neg A$, $\neg D$) makes S_1 unsatisfiable, thus we can proceed to the next branch. Figure 4.10(c) shows the next leaf node generated; it also makes S_1 unsatisfiable. Figure 4.10(d) illustrates how the algorithm backtracks up the search tree and then moves down the tree to locate another node, labeled ($\neg B$, $\neg D$, $\neg A$), which makes S_1 unsatisfiable. Since literal $\neg D$ plays no role in making S_1 unsatisfiable at this point, we reject the clause to which it belongs, i.e., the second positive cycle. Figure 4.10(e) shows the search tree after rejecting the clause corresponding to the second positive cycle, the result is a smaller tree to explore. Observe that the second positive cycle can be used later while exploring other branches of the tree. Figure 4.10(f) shows the complete search tree. Every leaf node makes S_1 unsatisfiable and there are no more branches to explore. Therefore, we can conclude unsatisfiability. □



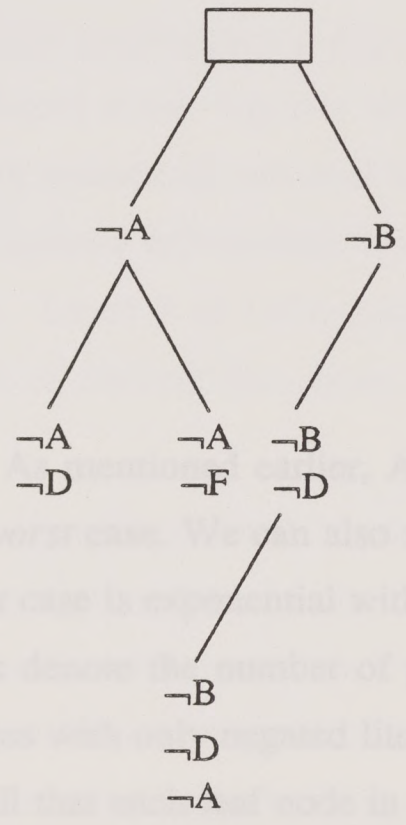
(a)



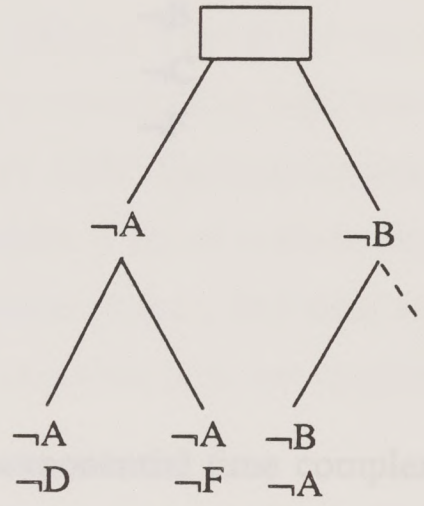
(b)



(c)

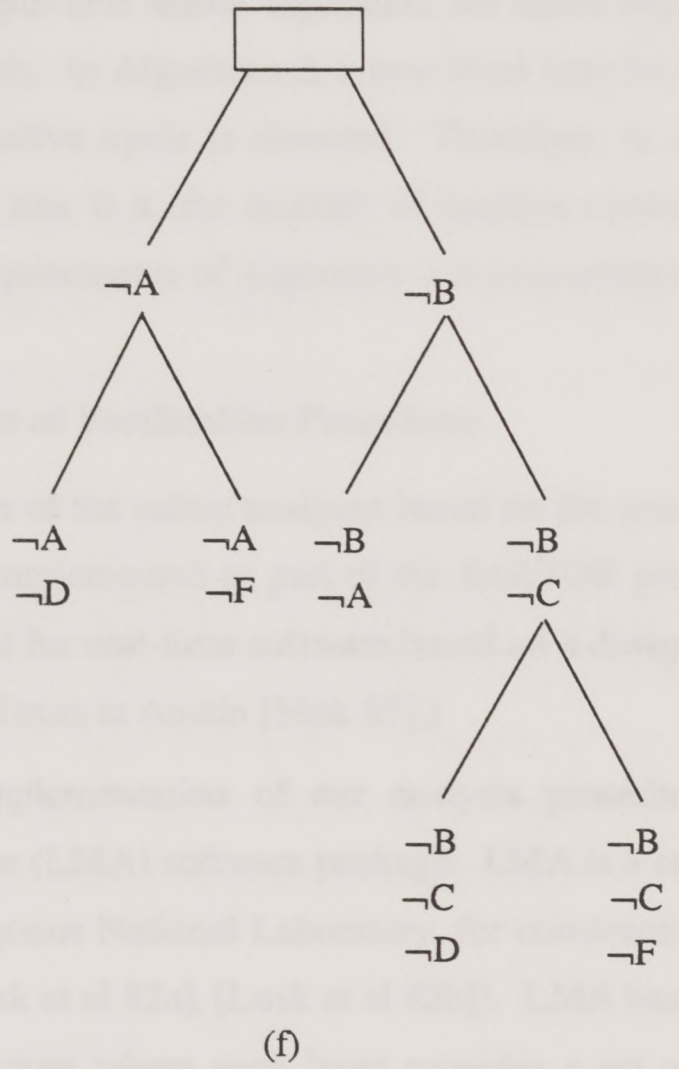


(d)



(e)

FIGURE 4.10



(f)

FIGURE 4.10

As mentioned earlier, Algorithm 3 has an exponential time complexity in the *worst* case. We can also show that the running time of this algorithm in the *worst* case is exponential with respect to the number of positive cycles detected. Let k denote the number of positive cycles detected. Let S_2 denote the set of clauses with only negated literals. (Each clause corresponds to a positive cycle.) Recall that each leaf node in a worst case search tree corresponds to a conjunction in the disjunctive normal form of the clauses in S_2. Hence, the running time of Algorithm 3 is proportional to the number of conjunctions in the disjunctive normal form of the clauses in S_2. For example, if S_2 contains two-literal clauses, there are 2^k leaf nodes (or conjunctions) in the *worst* case. Therefore, the running time is proportional to 2^k.

As with any depth-first search algorithm, the space requirement is proportional to the tree depth. In Algorithm 3, a new level may be added to the search tree each time a positive cycle is detected. Therefore, in the *worst* case, the height of the search tree is k, the number of positive cycles detected. Consequently, the space requirements of Algorithm 3 is proportional to k.

4.6. Implementation of Verification Procedure

The first version of the safety analyzer based on the procedure presented in this paper has been implemented as part of the SARTOR project. (SARTOR is a design environment for real-time software based on a design theory developed at the University of Texas at Austin [Mok 85].)

The current implementation of our analysis procedure uses the Logic Machine Architecture (LMA) software package. LMA is a set of software tools, developed at the Argonne National Laboratory, for constructing logic-based reasoning systems ([Lusk et al 82a], [Lusk et al 82b]). LMA has been implemented as a layered architecture where each layer provides a set of well-defined services. Layer 2 of LMA package, where list, clause, literal, and term are provided as abstract data types, was of particular importance in our implementation.

Algorithms 1 and 2 (graph construction and node removal operation) in our analysis procedure do not directly involve the notions of clause and literal. Hence, the two algorithms were implemented almost entirely independent of the LMA software. Algorithm 3, the unsatisfiability prover, treats positive cycles as clauses added to the existing set, and manipulates them to obtain contradictions. Hence, the layer 2 abstract data types of LMA were very useful in implementing Algorithm 3. In particular, clauses, literals, and terms are maintained in the internal object representaion supported by LMA; the implementation also relies extensively on the available procedures for their manipulation.

The current implementation, excluding the LMA software, consists of over 4000 lines of code written in the C programming Language.

4.7. Example: Nuclear Reactor Problem Revisited

This section describes a complete example illustrating the mechanical verification of a safety assertion using the procedure described in this chapter.

4.7.1. Specification

Recall the example in Section 2.3 on the system responsible for moving the control rods in a nuclear reactor. Three asynchronous components are involved in moving the control rods: Subsystem 1, Subsystem 2, and Manager. The specification of this system can be expressed in terms of RTL formulas which are in the subclass of formulas introduced in this chapter.

#

specification of subsystem 1

#

$\forall x1\ @(\Omega BUTTON1 , x1) \leq @(\uparrow RA1 , x1)$

$\forall x2\ @(\uparrow RA1 , x2) + 40 \leq @(\downarrow RA1 , x2)$

$\forall x3\ @(\downarrow RA1 , x3) \leq @((GRANT1{:=}T) , x3)$

$\forall x4\ @((GRANT1{:=}T) , x4) \leq @(\uparrow MOVE1 , x4) \wedge$

$@(\uparrow MOVE1 , x4) - 5 \leq @((GRANT1{:=}T) , x4)$

$\forall x6\ @(\downarrow MOVE1 , x6) - 20 \leq (\uparrow MOVE1 , x6)$

#

specification of subsystem 2

#

$\forall x7\ @(\Omega BUTTON2 , x7) \leq @(\uparrow RA2 , x7)$

$\forall x8\ @(\uparrow RA2 , x8) + 40 \leq @(\downarrow RA2 , x8)$

---

† Certain RTL axioms, as described in chapter 2, are omitted here since they do not play a role in the verification of this example.

$\forall$x9 @($\downarrow$RA2 , x9) $\leq$ @((GRANT2:=T) , x9)

$\forall$x10 @((GRANT2:=T) , x10) $\leq$ @($\uparrow$MOVE2 , x10)

@($\uparrow$MOVE2 , x10) - 5 $\leq$ @((GRANT2:=T) , x10)

$\forall$x12 @($\downarrow$MOVE2 , x12) - 20 $\leq$ ($\uparrow$MOVE2 , x12)

#

specification of manager

#

$\forall$x5 @((GRANT1:=T) , x5) + 30 $\leq$ @((GRANT1:=F) , x5)

$\forall$x11 @((GRANT2:=T) , x11) + 30 $\leq$ @((GRANT2:=F) , x11)

$\forall$x13 $\forall$x14 @((GRANT1:=F) , x13) $\leq$ @((GRANT2:=T) , x14) $\vee$

@((GRANT2:=F), x14) $\leq$ @((GRANT1:=T), x13)

To simplify, we list below the physical interpretation of the events which appeared in the RTL specification above.

ΩBUTTON1	: external event denoting pushbutton #1 is pressed
$\uparrow$RA1	: start of action to request access by Subsystem 1
$\downarrow$RA1	: stop of action to request access by Subsystem 1
GRANT1:=T	: event denoting request by Subsystem 1 is granted
GRANT1:=F	: manager assumes the granted request is released
$\uparrow$MOVE1	: start of action to move control rod #1
$\downarrow$MOVE1	: stop of action to move control rod #1

The remaining events $\uparrow$RA2, $\downarrow$RA2, GRANT2:=T, GRANT2:=F, $\uparrow$MOVE2, and $\downarrow$MOVE2 have similar meanings.

4.7.2. Safety Assertion

As described in Section 2.3, the desired safety property is that the reactor rods should be moved one at a time. In other words, execution of the action to move rod #1 may not overlap with the execution of the action to move rod #2.

Safety Assertion in RTL:

$\forall$i $\forall$j @($\downarrow$MOVE2 , j) < @($\uparrow$MOVE1 , i) $\vee$

$$@(\downarrow MOVE1\ ,\ i) < @(\uparrow MOVE2\ ,\ j)$$

Negation of Safety Assertion in RTL:

$$\exists i \exists j\ @(\uparrow MOVE1\ ,\ i) \leq @(\downarrow MOVE2\ ,\ j) \wedge$$
$$@(\uparrow MOVE2\ ,\ j) \leq @(\downarrow MOVE1\ ,\ i)$$

4.7.3. Verification

RTL formulas describing the specification and the safety assertion in the example fall within the subclass of RTL formulas introduced in Section 4.1.1. This subsection gives a proof, based on the analysis procedure in this chapter, showing that the safety assertion is a theorem derivable from the system specification.

We start with a set of RTL formulas representing the specification and the negation of the safety assertion. Prior to performing the analysis, we must transform the RTL formulas to the corresponding formulas in Presburger Arithmetic with uninterpreted integer functions. We replace each occurrence function with a specific event constant as its first argument by an uninterpreted integer function. For instance, @(↑MOVE1 , i) is replaced by an integer function StartMOVE1(i) which returns the time of the i^{th} occurrence of the start of action MOVE1. (StartMOVE1 is merely a function name with i as its integer argument.)

After the above transformation, the formulas are put in clausal form, as shown below. Symbols x1 through x14 are variables; I and J are Skolem constants; the function symbols are easy to identify from the syntax.

```
#
#  specification of subsystem 1
#
BUTTON1(x1) ≤ StartRA1(x1)
```

StartRA1(x2) + 40 ≤ StopRA1(x2)

StopRA1(x3) ≤ GRANT1true(x3)

GRANT1true(x4) ≤ StartMOVE1(x4) ∧

StartMOVE1(x4) - 5 ≤ GRANT1true(x4)

StopMOVE1(x6) - 20 ≤ StartMOVE1(x6)

#

specification of subsystem 2

#

BUTTON2(x7) ≤ StartRA1(x7)

StartRA2(x8) + 40 ≤ StopRA2(x8)

StopRA2(x9) ≤ GRANT2true(x9)

GRANT2true(x10) ≤ StartMOVE2(x10)

StartMOVE2(x10) - 5 ≤ GRANT2true(x10)

StopMOVE2(x12) - 20 ≤ StartMOVE2(x12)

#

specification of manager

#

GRANT1true(x5) + 30 ≤ GRANT1false(x5)

GRANT2true(x11) + 30 ≤ GRANT2false(x11)

GRANT1false(x13) ≤ GRANT2true(x14) ∨

GRANT2false(x14) ≤ GRANT1true(x13)

#

negation safety assertion

#

StartMOVE1(I) ≤ StopMOVE2(J)

StartMOVE2(J) ≤ StopMOVE1(I)

The analysis procedure first puts the set of formulas in a graph representation as described in Algorithm 1. Figure 4.11 shows the corresponding graph.

Repeated application of the node removal operation, Algorithm 2, reveals several positive cycles in the original graph. Two of these cycles are labelled on the graph of Figure 4.11. The edges in the positive cycle 1 are labelled $t_1, t_2, \ldots, t_6$. The edges in positive cycle 2 are labelled $s_1, s_2, \ldots, s_6$. The inequalities corresponding to the set of edges involved in the two positive cycles are as follows.

Inequalities in positive cycle 1:

t_1: GRANT2true(J) ≤ StartMOVE2(J)

t_2: StartMOVE2(J) ≤ StopMOVE1(I)

t_3: StopMOVE1(I) - 20 ≤ StartMOVE1(I)

t_4: StartMOVE1(I) - 5 ≤ GRANT1true(I)

t_5: GRANT1true(I) + 30 ≤ GRANT1false(I)

t_6: GRANT1false(I) ≤ GRANT2true(J)

Inequalities in positive cycle 2:

s_1: GRANT1true(I) ≤ StartMOVE1(I)

s_1: StartMOVE1(I) ≤ StopMOVE2(J)

s_3; StopMOVE2(J) - 20 ≤ StartMOVE2(J)

s_4: StartMOVE2(J) - 5 ≤ GRANT2true(J)

s_5: GRANT2true(J) + 30 ≤ GRANT2false(J)

s_6: GRANT2false(J) ≤ GRANT1true(I)

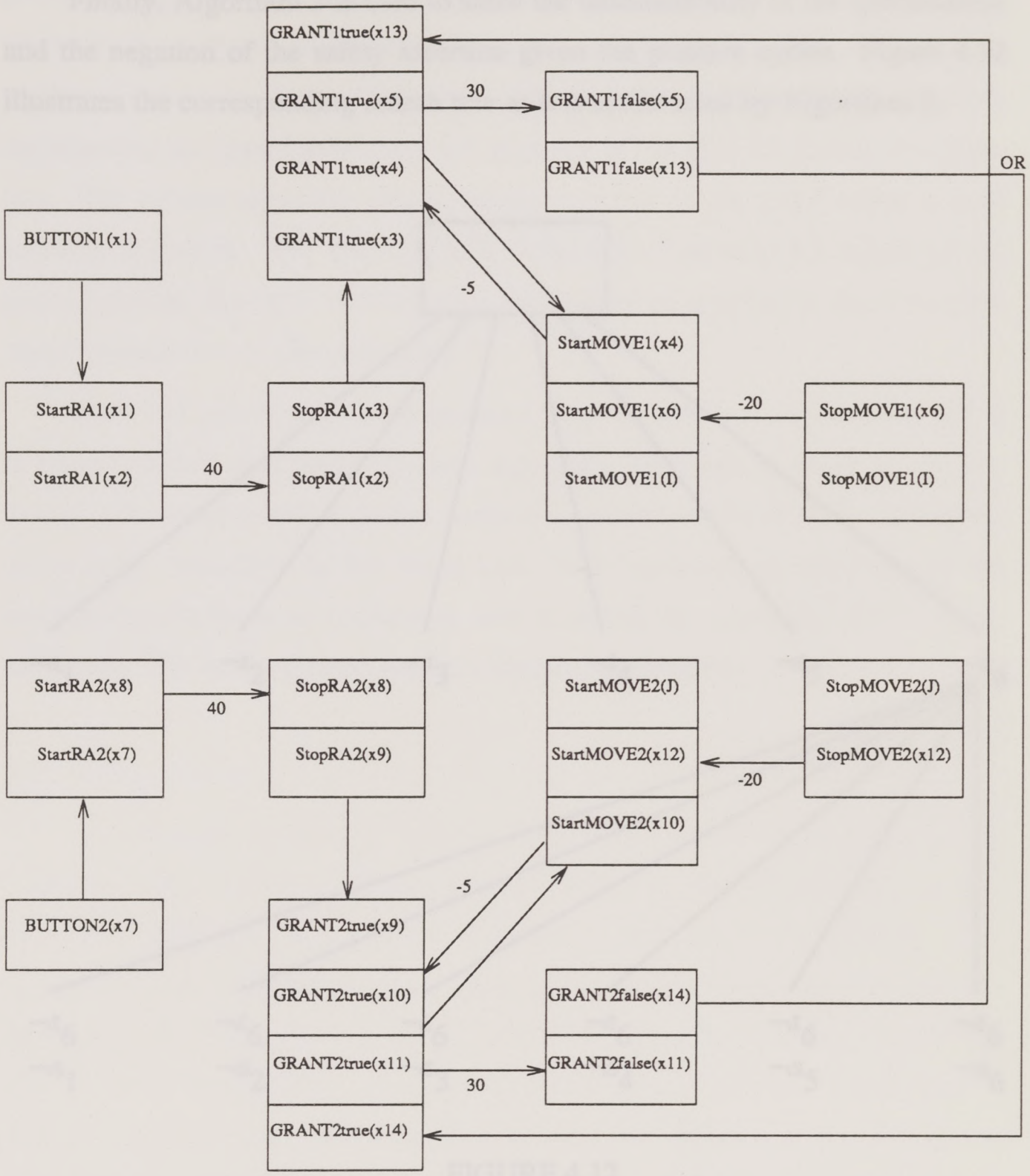

GRANT1true(x13)
GRANT1true(x5)
30
GRANT1false(x5)
GRANT1true(x4)
GRANT1false(x13)
OR
BUTTON1(x1)
GRANT1true(x3)
-5
StartMOVE1(x4)
StartRA1(x1)
StopRA1(x3)
StartMOVE1(x6)
-20
StopMOVE1(x6)
StartRA1(x2)
40
StopRA1(x2)
StartMOVE1(I)
StopMOVE1(I)
StartRA2(x8)
40
StopRA2(x8)
StartMOVE2(J)
StopMOVE2(J)
StartRA2(x7)
StopRA2(x9)
StartMOVE2(x12)
-20
StopMOVE2(x12)
StartMOVE2(x10)
-5
BUTTON2(x7)
GRANT2true(x9)
GRANT2true(x10)
GRANT2false(x14)
GRANT2true(x11)
30
GRANT2false(x11)
GRANT2true(x14)

FIGURE 4.11

Finally, Algorithm 3 is used to show the unsatisfiability of the specification and the negation of the safety assertion given the positive cycles. Figure 4.12 illustrates the corresponding search tree as it is constructed by Algorithm 3.

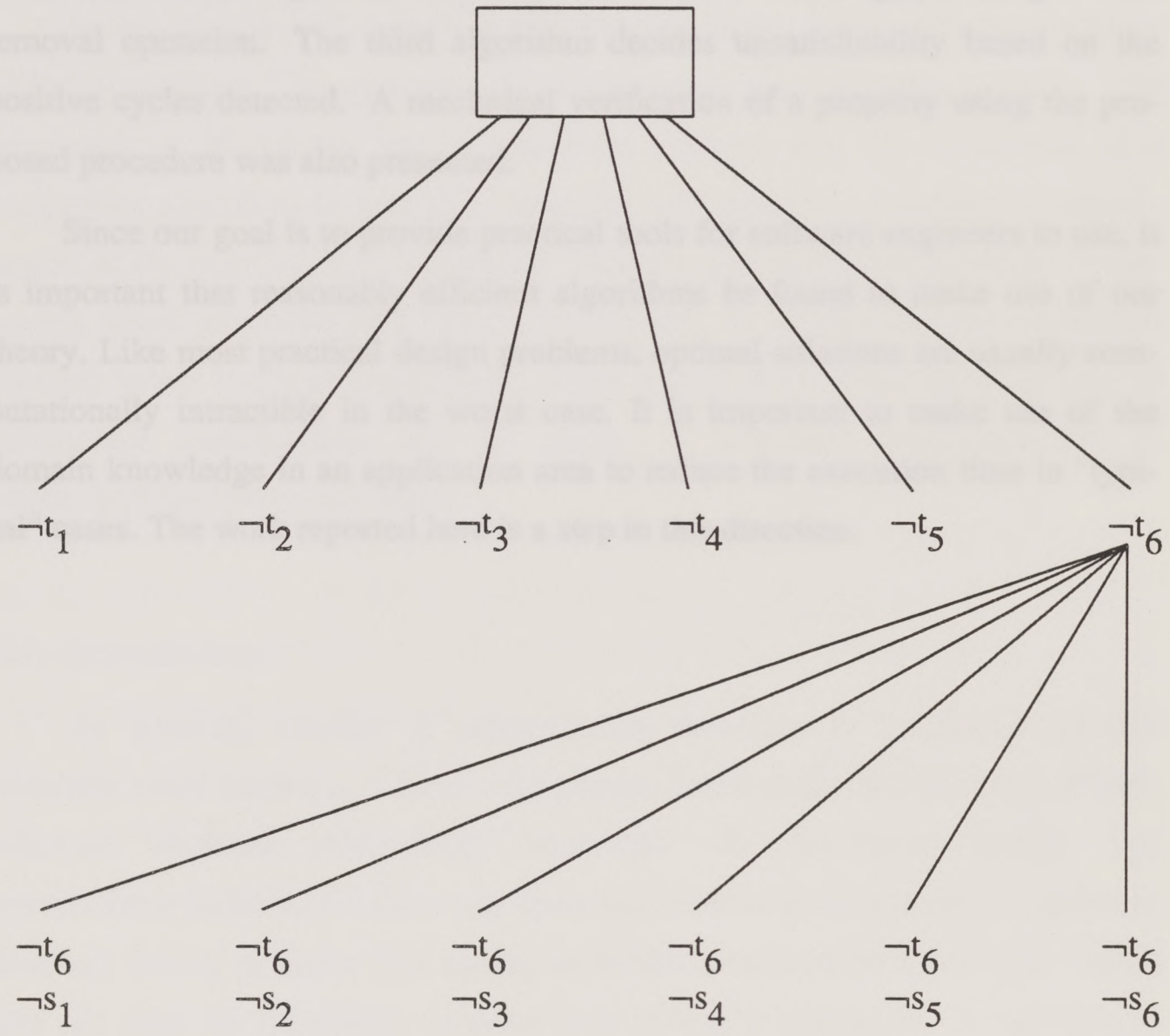


FIGURE 4.12

4.8. Concluding Remarks

This chapter presented a graph-theoretic procedure for verification of properties of real-time systems expressible in a subclass of Real Time Logic. To show that a timing property is consistent with a given specification, we need to

prove that the corresponding formula consisting of the specification in conjunction with the negation of the timing property is unsatisfiable. The analysis procedure is based on three algorithms. The first algorithm constructs a graph representing the specification and the negation of the property under investigation. The second algorithm detects positive cycles in the graph using a node removal operation. The third algorithm decides unsatisfiability based on the positive cycles detected. A mechnical verification of a property using the proposed procedure was also presented.

Since our goal is to provide practical tools for software engineers to use, it is important that reasonably efficient algorithms be found to make use of our theory. Like most practical design problems, optimal solutions are usually computationally intractible in the worst case. It is important to make use of the domain knowledge in an application area to reduce the execution time in "typical" cases. The work reported here is a step in this direction.

Chapter 5

Modechart: a Spec. Language for Real-Time Systems

In this chapter, we present a specification language for real-time systems called Modechart. The semantics of Modechart is given in terms of RTL. The semantics of Modechart has an important property that the translation of a Modechart specification into RTL formulas will result in a hierarchical organization of the resulting RTL assertions. This gives us significant leverage in reasoning about properties of a system by allowing us to filter out assertions that concern lower levels of abstraction. Some results about desirable properties of Modechart specifications will be given. A graphical implementation of Modechart has been completed.

5.1. Introduction

As software control of safety-critical functions in *embedded systems* becomes more common, a means for precise and concise specification of their behavior becomes increasingly important. As mentioned earlier, The specification problem for these real-time applications is more complex since the absolute timing behavior (the timing of events measured by a real-time clock) and not only the functional behavior of a system is important. In addition to being a valuable aid to the maintenance of complex real-time systems, a precise and concise specification language can also be used to provide answers to queries about system behavior without a full implemenation (rapid prototyping). In this chapter, we shall present a specification language for real-time systems called Modechart. The semantics of Modechart is defined in terms of Real Time Logic. A graphical implementation of Modechart has also been completed.

Substantial work has been done by a number of researchers in the specification and prototyping of complex real-time systems,
e.g.,[Zave 85], [Luqi & Berzins 87], [Alford 77], [Davis & Vick 77]. A notable experiment in the application of specification methods is the Software Cost Reduction project at the Naval Research Laboratory where a systematic methodology was applied to document the software requirements of the A-7E aircraft (see [Heninger 80], [Parnas et al 78]). More recently, Harel proposed a visual language called Statechart [Harel 86] which is an extension of conventional state-transition diagrams. Harel's language provides a succinct way to represent large systems since it supports hierarchical and modular decomposition of state machines. A formal semantics of Statechart in terms of temporal logic and a general theory of *reactive systems* are also being investigated by its inventors [Harel et al 87].

Unlike previous work, our work emphasizes the specification of absolute timing properties of systems. In Modechart, we make use of the concept of *modes* from the work of Parnas *et al* at the Naval Research Laboratory. Modes can be thought of as partitions of the state space of a system and are an effective way for modular specification of large state machines. Modechart also borrows from Statechart the very appealing compact representation of large state machines. Our main contribution is in providing a semantics which explicitly deals with the absolute timing of events and avoids some of the potential semantic anomalies of Statechart. More importantly, the translation of a Modechart specification into RTL formulas will result in a hierarchical organization of the resulting RTL assertions. This gives us significant leverage in reasoning about properties of a system by allowing us to filter out assertions that concern lower levels of abstraction. The ability to avoid considering all the assertions defining the behavior of a large system is a prerequisite to practical applications of verification technology.

Modechart has been developed as a graphical specification tool in SARTOR (Software Automation for Real-Time OpeRations), a design environment for hard-real-time software currently under development at the University of Texas at Austin. The goal of SARTOR is to mechanize the analysis and synthesis of real-time software from systems specification. An implementation of Modechart serves as a front-end for SARTOR which provides a suit of tools to analyze a specification for satisfaction of safety requirements. A design can be synthesized (rapid prototyping) if it is determined that there are sufficient resources to do so. A review of SARTOR can be found in[Mok 85].

The rest of this chapter is organized as follows. Section 5.2 gives the syntactic description of modes. Section 5.3 describes the RTL semantics of mode transitions. Section 5.4 gives the RTL semantics of actions in modes. Section 5.5 deals with mode transitions that span more than one level of the mode hierarchy. Section 5.6 presents some results about desirable properties of mode specifications using Modechart. section 5.7 is the conclusion.

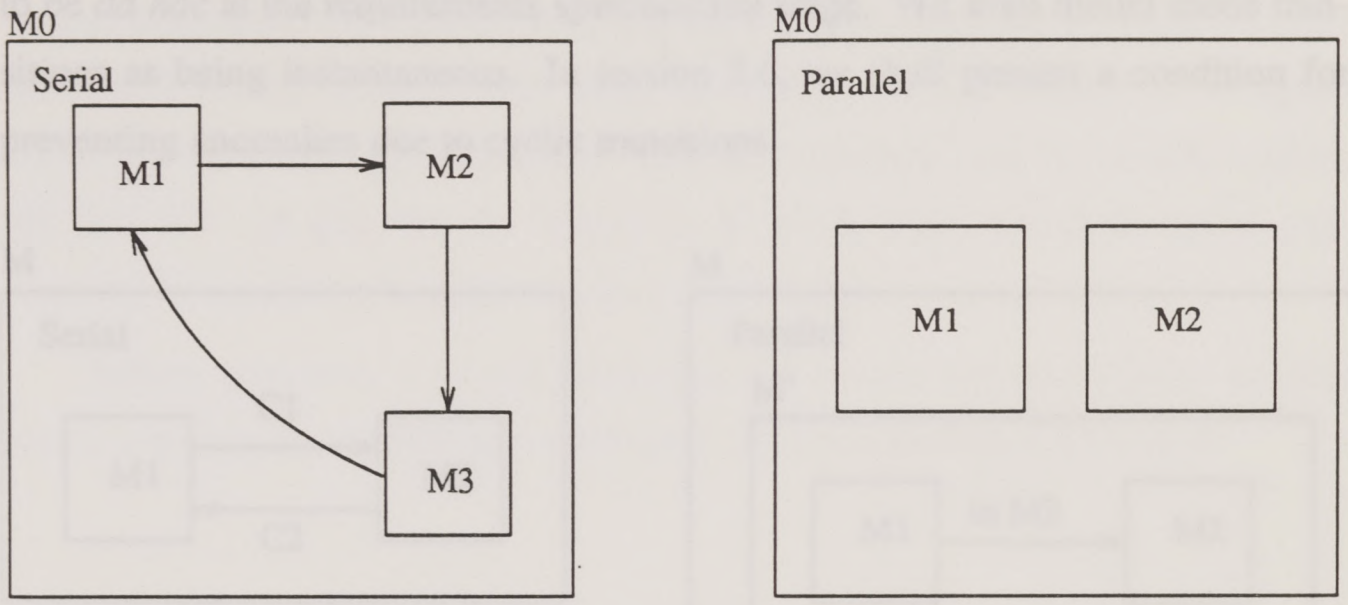
5.2. Hierarchical Decomposition: Parallel and Serial Modes

Intuitively, modes may be viewed as control information that impose structure on the operation of a system. Modes are arranged hierarchically. Furthermore, modes which are peers in this hierarchy can be related in one of two ways: in series or in parallel.

The series relationship among several modes indicates that the system operates in, at most, one of these modes at any time. For instance, Figure 5.1(a) illustrates a two-level hierarchy where M0 is the parent mode and M1, M2, and M3 are embedded in M0. If the system is in mode M0, then it must also be in either one of M1, M2 or M3. M0 is said to be a *serial* mode, and M1, M2, and M3 are said to be *in series*. A transition allows the system to go from one mode to another. Since a specification may require a transition to go across different

levels of the hierarchy, transitions may not always be between modes at the same level in a hierarchy. Also, entry into a serial mode M requires the designation of one of the child modes of M to be the default mode the system will be in. If the transition arrow crosses into the serial mode, then the child mode that the transition arrow points at is entered. Otherwise, a child mode which is labeled as the *initial* mode is entered. As an example, mode M0 of Figure 5.1(a) could model the flight path monitoring mechanism for an aircraft. Modes M1, M2, and M3 represent *Monitor*, *Signal*, and *Correction* modes, respectively. When the plane is detected to be off course, the monitor mechanism moves from the Monitor mode into the Signal Mode. In the signal mode, the pilot is warned that the plane is off course. The pilot corrects the flight of the aircraft, taking the system into the Correction mode. After making the correction, the pilot informs the system (e.g., by pressing a button), returning the system to the monitor mode. Monitor, Signal, and Correction modes are in series because the monitor mechanism can only be in one of the three at a time.

The parallel relationship among several modes indicates that a system operates in all of these modes simultaneously. For instance, figure 5.1(b) illustrates a mode M0 with two embedded modes, M1 and M2. If the system is in mode M0, then it also must be in both M1 and M2. (M0 is said to be a *parallel* mode, and M1 and M2 are said to be *in parallel*.) Transitions between modes in parallel are not allowed. Entry into a parallel mode requires entry into all of its immediate child modes. Hence, no member of a set of modes in parallel should be specified as the initial mode. Similarly, a transition out of one mode requires exit out of all the modes in parallel to it. Returning to the aircraft example, mode M0 of Figure 5.1(b) could model the plane's instrument display system. Modes M1 and M2 might represent the *Airspeed* and *Altitude* display modes, respectively. The airspeed and altitude displays are updated independently of each other and so are in parallel under mode M0.



(a) Serial relationship

(b) Parallel relationship.

FIGURE 5.1

The preceding informal description of serial and parallel modes might give a deceptively simple view of Modechart. In fact, there are many ambiguities which must be resolved in providing a formal semantics for mode transitions. For example, if transition between two modes is instantaneous, then a cyclic sequence of transitions occuring at the same instant of time will lead to an anomaly. Consider the serial mode M in figure 5.2(a). If the system is in mode M1 and condition C1 is true at time t, then the corresponding transition is taken and the system exits from mode M1 and enters mode M2 at that instant of time. However, if condition C2 is also true at time t, then the system can exit M2 and enter M1 at the same instant of time. Hence, the modes in the cycle would be entered and exited an infinite number of times at an instant in time. To avoid this anomaly, one might require mode transitions to take finite time. The problem is that system behavior will then be ambiguous when an event occurs in between modes and the response to that event is supposed to depend on which mode the system is in, and any bound put on the duration of a transition is likely

to be *ad hoc* at the requirements specification stage. We shall model mode transitions as being instantaneous. In section 5.6, we shall present a condition for preventing anomalies due to cyclic transitions.

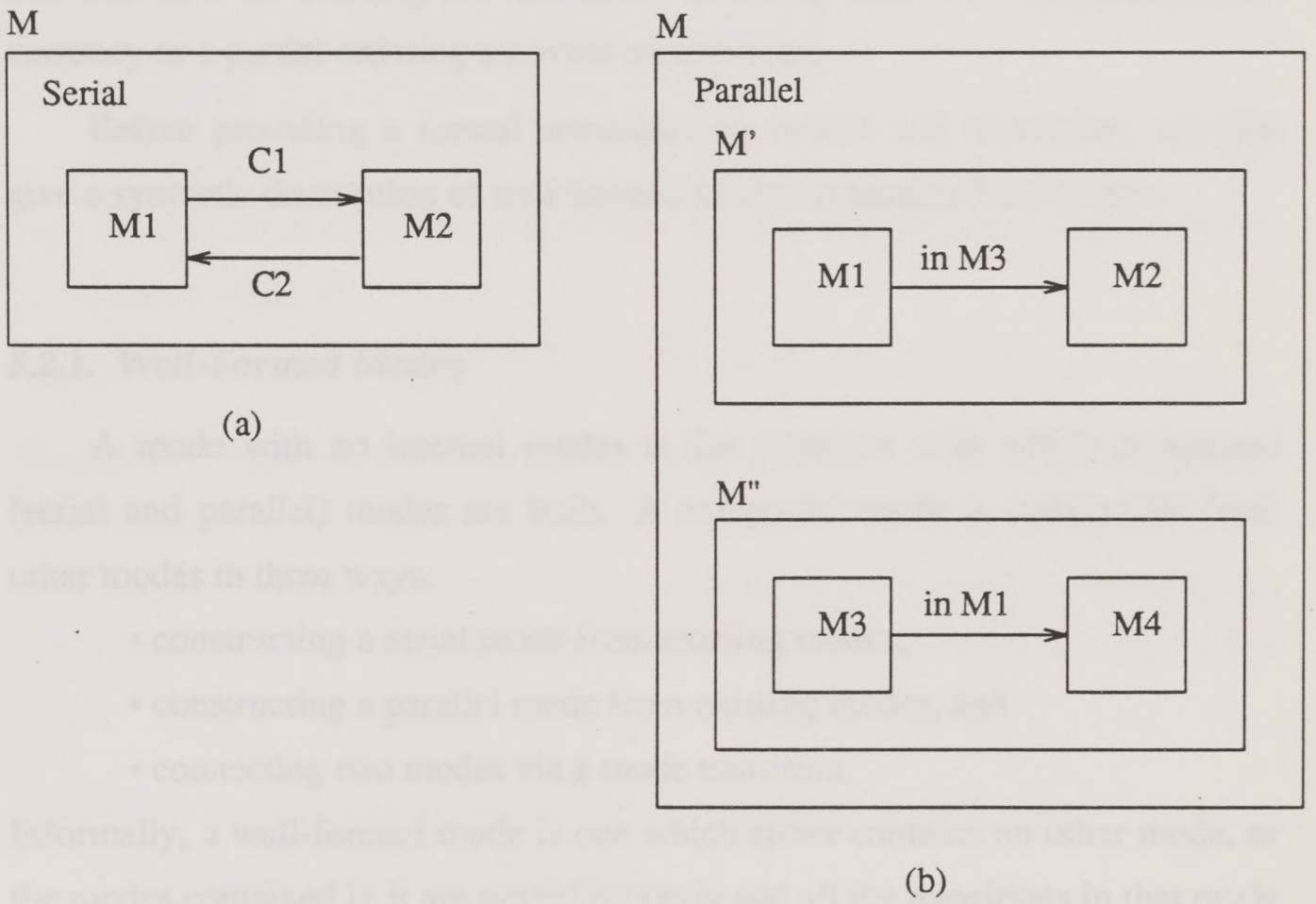


FIGURE 5.2

Ambiguities can also arise due to the way one may model concurrency. For example, consider the parallel mode M in figure 5.2(b). The system is in modes M1 and M3 initially. If the behavior of a system specified in Modechart is viewed as a sequence of non-overlapping events (i.e., following the interleaving model of concurrency), then the system in figure 5.2(b) cannot end up in both modes M2 and M4 even though one may expect simultaneous transitions from M1 to M2 and from M3 to M4 to take the system into modes M2 and M4. This is due to the fact that two simultaneous events are regarded as equivalent to the occurrence of the two events in either order.† However, if concurrency in a

system is modeled by a partial ordering among events (i.e., two events are concurrenct if neither precedes the other one), then the two transitions in figure 5.2(b) can be taken simultaneously. Hence, the system can be in both modes, M2 and M4. In defining the semantics of Modechart, we shall model concurrency as a partial ordering on event occurrences.

Before providing a formal semantics for modes and transitions, we first give a syntactic description of well-formed modes in section 5.2.1 below.

5.2.1. Well-Formed Modes

A mode with no internal modes is the primitive with which compound (serial and parallel) modes are built. A compound mode is constructed from other modes in three ways:

- constructing a serial mode from existing modes,
- constructing a parallel mode from existing modes, and
- connecting two modes via a mode transition.

Informally, a well-formed mode is one which either contains no other mode, or the modes contained in it are nested properly and all the transitions in that mode do not connect parallel modes. Furthermore, a well-formed serial mode may have at most one initial mode and a well-formed parallel mode may not have an initial mode. It should be noted that well-formedness is a syntactical property which by itself does not guarantee that all initial modes are properly designated. A well-formed mode may still be ambiguous in the sense that its initial mode designations may be underspecified. We shall return to this point shortly.

Definition: *Containment Partial Order*

Let $\boldsymbol{M}$ be a set of modes $\{M_1, M_2, \cdots, M_m\}$. The binary relation *containment*, denoted by $\subset$, defines a partial ordering of the elements of the set

---

† In this example, it is possible to allow the system to end up in both M2 and M4 in the interleaving model by stipulating additional causality rules governing events that occur "at the same time". The semantics of this approach is rather complicated [Pnueli 87].

M. Graphically, $M_i \subset M_j$ iff the box representing mode M_i is contained entirely inside the box representing mode M_j.

Definition:

A mode M_i is a *primitive* mode iff $\forall j \ M_j \not\subset M_i$.

A mode M_i is a *compound* mode iff $\exists j \ M_j \subset M_i$.

A mode M_i is a *root* mode iff $\forall j \ M_i \not\subset M_j$.

A mode M_i is an immediate *child* of a mode M_j (and conversly M_j is the *parent* of M_i) iff

$$M_i \subset M_j \ \wedge \ (\forall k \ j \neq k \wedge M_i \subset M_k \ \rightarrow \ M_j \subset M_k)$$

Graphically, a primitive mode contains no other mode; a compound mode contains at least one other mode; a root mode is one which is not contained in any other mode. Furthermore, the box representing a child mode is immediately surronded by the box representing the parent. We now define a well-formed mode.

Definition: *well-formed* modes are defined recursively as follows:

1. A primitive mode is well-formed.

2. Suppose $M_1, M_2, \cdots, M_m$ are well-formed root modes, and at most one mode M_i is labelled as the initial mode, then adding a serial mode M consisting of only these modes as its immediate children makes M well-formed.

3. Suppose $M_1, M_2, \cdots, M_m$ are well-formed root modes, and none is labelled as an initial mode, then adding a parallel mode M consisting of only these modes as its immediate children makes M well-formed.

4. Let M be a well-formed mode, and $M_i \subset M$ and $M_j \subset M$. Adding a transition from M_i to M_j preserves M as a well-formed mode if the least upper

bound of M_i and M_j with respect to the $\subset$ relation is a *serial* mode.†

Informally, the above condition requires that the first mode containing both M_i and M_j to be a serial mode. Observe that the least upper bound of the two modes is either M or a mode inside M. Also notice that it is not necessary for the children of a serial mode to be connected by any transition, since they can be entered by distinct transitions from outside the parent. None of the children in this case needs to be designated an initial mode.

As mentioned earlier, a well-formed mode can still be ambiguous if designation of the initial modes is underspecified. As an example, consider the case where a transition enters a mode in parallel to a serial mode M, but none of the immediate children of M is labeled as the initial mode. We first present a definition which is useful in formulating a condition for ensuring proper designation of the initial modes.

Definition: A mode M is a *landing* mode if any one of the following conditions holds:

- if M is the root mode,
- if a transition ends at mode M,
- if its parent M' is a serial mode,
  M' is a landing mode, and
  M is the initial mode,
- if its parent M' is a parallel mode, and
  M' is a landing mode
- if its parent M' is a parallel mode, and

---

† The least upper bound of M_i and M_j is a mode M' such that

$M_i \subset M' \wedge M_j \subset M' \wedge$ (for each mode M'' immediate child of M', $M_i \not\subset M'' \vee M_j \not\subset M''$)

there is a transition ending at or crossing
into a mode in parallel with *M*.

In a system whose root mode is well-formed, we define the UDIM condition (for Unambiguous Designation of Initial Modes) as follows:

UDIM Condition: For each serial mode which is a landing mode, there is exactly one initial mode.

To illustrate the above condition, assume that the system enters a serial mode M. If M is not a landing mode, the only way to enter M is to have the corresponding transition arrow cross into it. In this case, the transition identifies a unique child of M to be entered, i.e., there is no ambiguity. However, if M is a landing mode, the UDIM condition requires M to have an initial mode, thus ensuring against ambiguity.

In the subsequent three sections, the formal semantics of Modechart is presented in terms of RTL formulas.

5.3. Specifying the Semantics of Modes and Transitions in RTL

In this section, we introduce additional classes of events to help capture the formal semantics of modes in RTL. After presenting the syntax for the conditions on mode transitions, we shall illustrate how mode transitions are specified in RTL.

5.3.1. Comparison of Modes and State Variables

Since modes represent control information about the behavior of a system, it may seem natural to use state variables to model modes in RTL. In fact, we use a notation similar to state variables to represent modes in RTL. The semantic distinctions between modes and state variables will be discussed shortly.

To capture the formal semantics of modes in RTL, we introduce two events denoting mode entry and exit. For a mode M, the event (M:=T) denotes entering M and the event (M:=F) represents exiting the mode. (To distinguish these events from transition events for state variables, we shall refer to them as the *mode entry* and the *mode exit* events.) Staying in a mode during an interval is described through the use of a notation similar to state predicates. The notation, refered to as the *mode predicate*, also has nine variations for each mode M:

M[x,y], M[x,y), M[x,y>, M(x,y), M(x,y], M(x,y>,

M<x,y>, M<x,y], and M<x,y).

Each mode predicate qualifies the timing of two events, one denoting the entry to a mode and the other denoting the exit from a mode. The two arguments, x and y, in a mode predicate are used in conjunction with the symbols "[", "]", "(", ")", "<", and ">" to denote an interval over which the system remains in a mode.

The convention we use for arriving at this syntax requires some explanation. Suppose E and E' denote the mode entry and the mode exit events for a mode M, respectively. Informally,

"[x" denotes that E occurs at time x,

"(x" denotes that E occurs before or at time x,

"<x" denotes that E occurs before time x,

"y]" denotes that E' occurs at time y,

"y)" denotes that E' does not occur before time y,

"y>" denotes that E' does not occur before or at time y.

For example, the mode predicate M[x,y] indicates that the system enters mode M at time x and exits mode M at time y. Precisely, the mode entry event (M:=T) occurs at time x, the system remains in mode M during the interval between x and y, and the mode exit event (M:=F) occurs at time y. For another example, M[x,y) indicates that the system enters mode M at time x and it remains in this mode at least until time y. We define the notation M(x,x) to

denote that the system is *in mode* M at time x. Due to the above definition, when a transition from a mode M1 to another mode M2 is taken, the system is *in* both modes at that instant of time. However, as it will be shown in section 5.6, the semantics of Modechart prevents the system to be in two modes in series over a finite time interval. We stress that mode predicates are formally defined in terms of the corresponding mode entry and exit events.

Despite the similarities in their notations, there are important semantic distinctions between modes and state variables. Intuitively, a state variable represents information about data whereas a mode represents some control information about the system. The value of a state variable is changed *explicitly* by completing the execution of an action which takes non-zero units of time to perform. Consequently, a state attribute S cannot become true and then become false at the same instant of time, i.e., the two events (S:=T) and (S:=F) cannot happen at the same instant of time. A mode entry or exit is *implicit* in that it does not require the execution of an action; a mode transition is taken when a certain condition is satisfied. Hence, a mode M can be entered and exited simultanesouly if the condition for exit is also satisfied, i.e., we allow the two events (M:=T) and (M:=F) to happen at the same instant of time.

5.3.2. Transitions

Transition between two modes represents a change in the control information of the system. A mode transition is an instantaneous event which takes zero time units. To capture the formal semantics of a transition, we introduce the *mode transition event* to denote the occurrence of a transition from one mode to another. Specifically, we use the notation $(M_i\text{-}M_j)$ to indicate the mode transition event from mode M_i to mode M_j.

We associate a condition with each transition. The condition for a mode transition is of the form

$$c_1 \vee c_2 \vee \cdots \vee c_n$$

where each disjunct c_k is either (1) a triggering condition for taking the transition, or (2) a lower/upper bound restriction on when the transition may be taken.

(1) Triggering Condition: A disjunct c_k denoting a triggering condition is of the form

$$p_1 \wedge p_2 \wedge \cdots \wedge p_m$$

where the $p_j s$ specify a condition for taking the transition depending on the occurrence of an event and/or the truth values of certain predicates. In particular, each subcondition p_j is of one of the three forms shown below. (E denotes an event, S is a state variable and t denotes the time at which the transition is taken.)

(a) S (or $\overline{S}$)

Enabling subcondition is a state variable S being true (or false) at time t.

(b) $\{M_1, M_2, \cdots, M_m\}$

Enabling subcondition is the system being in at least one of the specified modes at a finite interval upto time t.

(c) E

Enabling subcondition is the occurrence of an event E at time t where E can be an

- external event, e.g., ΩE,
- event denoting start of an action, e.g., ↑A,
- event denoting completion of an action, e.g., ↓A,
- event setting a state variable to true, e.g., (S:=T),
- event setting a state variable to false, e.g., (S:=F),
- event denoting entry into mode, e.g., ($M1$:=T), or
- an event denoting exit from mode, e.g., ($M1$:=F).

A transition from a mode is taken at a time t iff all the subconditions are satisfied at time t when the system is in that mode. Each of the three forms of a subcondition p_j can be expressed as an RTL predicate:

(a) $S(t,t)$ (or $\overline{S}(t,t)$)

(b) $M_1<t,t) \vee M_2<t,t) \vee \cdots \vee M_m<t,t)$

(c) $@(E,\mathrm{i}) = \mathrm{t}$

In the above, t is the time at which the triggering condition for the transition holds, i.e., the transition is taken at that instant of time. We will shortly illustrate how these predicates are used in RTL formulas to express a mode transition.

(2) Lower/Upper Bound Condition: A condition c_k denoting a lower/upper bound restriction is of the form

(r,d)

where r is a non-negative integer denoting a delay and d is a positive integer or ∞ denoting a deadline. If a lower/upper bound is specified as the condition for a transition from a mode, the transition can be taken after r time units and before d time units has elapsed since entering the mode. Three special forms of lower/upper bound condition are of particular interest. The condition

alarm r

is used to represent the case where the delay and the deadline on a transition are equal. In this case, the transition is taken exactly after r time units has elapsed since entering the mode. The condition

delay r

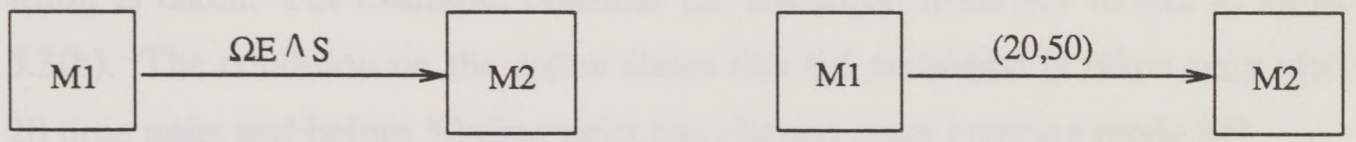
specifies a lower bound on when the transition can be taken without specifying an upper bound, i.e., the condition (r,∞). The condition

deadline d

specifies a finite upper bound and no lower bound on the transition, i.e., the condition (0,d). The lower/upper bound restriction imposed on a transition from a mode M can be expressed as RTL predicates:

$$t + r \leq t' \wedge t' \leq t + d$$

where t is the time at which mode M is entered and t' is the time at which the transition is taken.



(a) (b)

FIGURE 5.3

Having discussed the two forms of conditions on mode transitions, the triggering condition and lower/upper bound condition, we can show how a mode transition is formally specified in RTL. Suppose there is a transition from mode M1 to mode M2 and this transition is the only one out of mode M1. A triggering condition c_k on the transition arrow from mode M1 to mode M2 is captured by the following RTL formula:

$$\forall t \ \text{M1}(t,t) \wedge C \rightarrow \exists j \ @((\text{M1-M2}),j) = t$$

where (M1-M2) is the mode transition event fom M1 to M2, C is the conjunction of RTL predicates specifying the subconditions (the $p_j s$) in c_k, and t is the time at which the transition is taken. As an example, consider the two modes M1 and M2 in Figure 5.3(a). The condition on the arrow states: while in mode M1, if the external event ΩE occurs when state variable S is true, the transition from mode M1 to mode M2 is taken. The corresponding RTL formula representing this mode transition is as follows:

$$\forall t \forall i \; M1(t,t) \wedge @(\Omega E,i)=t \wedge S(t,t) \rightarrow \exists j \; @((M1\text{-}M2),j) = t$$

A lower/upper bound condition c_k on a transition arrow from M1 to M2 is expressed by the following RTL formula:

$$\forall t \; M1[t,t) \rightarrow \exists t' \exists j \; @((M1\text{-}M2),j) = t' \wedge B$$

where (M1-M2) is the mode transition event from M1 to M2, B is the conjunction of of RTL predicates specifying the lower/upper bound restrictions on the transition, t is the time of entering mode M1, and t' is the time at which the transition is taken. For example, consider the transition from M1 to M2 in figure 5.3(b). The condition on the arrow states that the transition is taken only after 20 time units and before 50 time units has elapsed since entering mode M1.

$$\forall t \; M1[t,t) \rightarrow \exists t' \exists j \; @((M1\text{-}M2),j) = t' \wedge t + 20 \leq t' \leq t + 50$$

In the preceding formulas, it is assumed that the transition from M1 to M2 is the only way to exit M1. In the case of multiple transitions and nested serial/parallel modes, a transition out of a mode is captured by a larger formula which is composed of formulas like the ones shown above. This will be dealt with in section 5.5.

5.4. Actions and Timing Constraints

A real-time system may be required to execute certain actions while operating in some mode, or upon transition from one mode to another. Furthermore, the initiation and completion of actions are often subject to timing constraints. This section discusses the RTL semantics of actions in relation to mode transitions.

5.4.1. Actions Upon Mode Transitions

A system designer may wish to specify an action to be performed when an event triggering a mode transition occurs. For example, an action may be required to set a state variable upon exiting a mode and entering another, as shown in the system in figure 5.4. In this system, there are two modes in series: MAGNET-ON and MAGNET-OFF. The transitions from MAGNET-ON to MAGNET-OFF happens when pushbutton #1 is pressed (external event ΩPRESS1); the transition from MAGNET-OFF to MAGNET-ON happens when pushbutton #1 is released (external event ΩRELEASE1). The state variable MAG must be set to false when ΩPRESS1 (triggering one transition) occurs and set to true when ΩRELEASE1 (triggering the other transition) occurs. We refer to the above two actions as A and B, respectively.

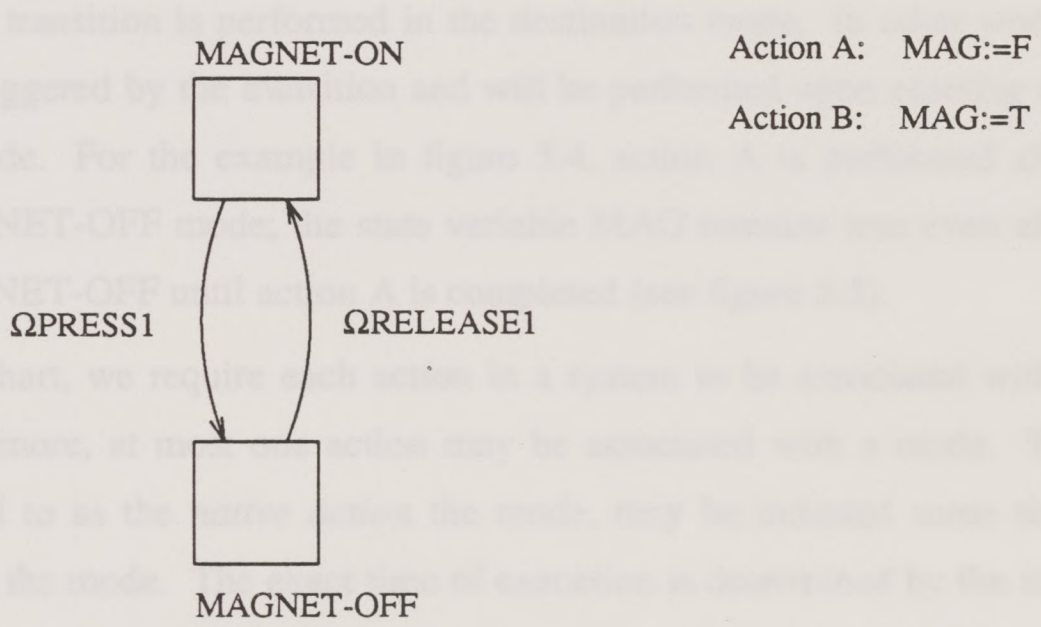


FIGURE 5.4

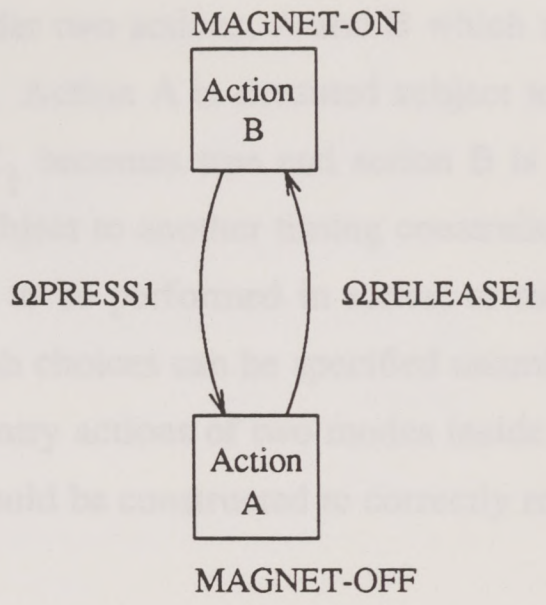
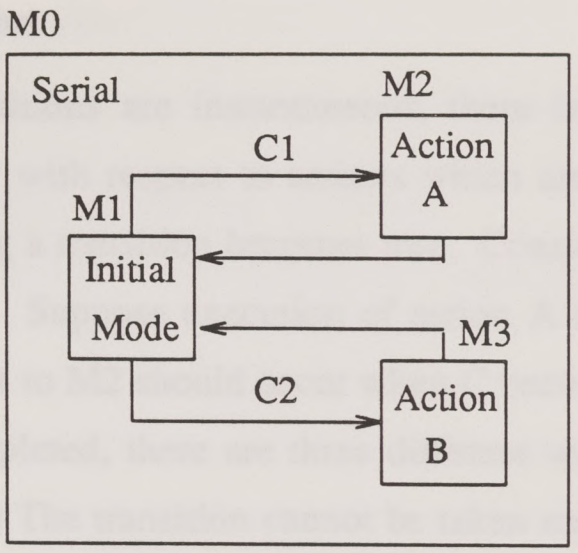


FIGURE 5.5

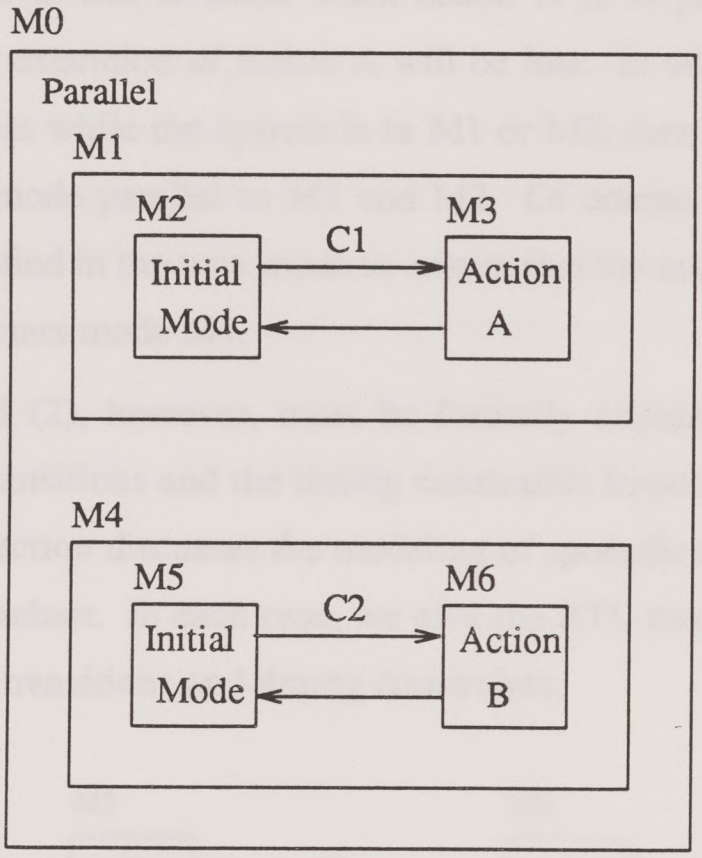
Since mode transitions are instantaneous in Modechart, an action to be performed upon a transition is performed in the destination mode. In other words, the action is triggered by the transition and will be performed upon entering the destination mode. For the example in figure 5.4, action A is performed after entering MAGNET-OFF mode; the state variable MAG remains true even after entering MAGNET-OFF until action A is completed (see figure 5.5).

In Modechart, we require each action in a system to be associated with a mode. Furthermore, at most one action may be associated with a mode. The action, referred to as the *native action* the mode, may be initiated some time after entry into the mode. The exact time of execution is determined by the timing constraint imposed on the action. (In this paper, a timing constraint on an action is either *sporadic* or *periodic*.) The requirement of at most one entry action per mode may seem overly restrictive, since two or more actions with different timing constraints may need to be performed when a system is in a certain mode. Our approach is to create a child mode for each action. This way, no additional mechanism needs to be introduced in case execution of actions in the same mode is also subject to precedence or mutual exclusion constraints.

For example, consider two actions, A and B which may be executed when a system is in mode M0. Action A is executed subject to a certain timing constraint when condition C_1 becomes true and action B is executed when condition C_2 becomes true, subject to another timing constraint. An application may require actions A and B to be performed in series, or the two actions must be executed in parallel. Both choices can be specified unambiguously by designating actions A and B as entry actions of two modes inside mode M0. Figure 5.6 shows how mode M0 should be constructed to correctly reflect each choice:


M0
Serial
M2
C1
Action
A
M1
Initial
Mode
M3
C2
Action
B

(a)


M0
Parallel
M1
M2
C1
M3
Initial
Mode
Action
A
M4
M5
C2
M6
Initial
Mode
Action
B

(b)

FIGURE 5.6

5.4.2. Actions in Modes

Since mode transitions are instantaneous, there is a need to be precise about system behavior with respect to actions which are being executed when the condition for taking a transition becomes true. Consider the two modes, M1 and M2 shown below. Suppose execution of action A is started in mode M1. The transition from M1 to M2 should occur when C becomes true. If the execution of A has not completed, there are three different ways the execution of A may be terminated: (1) The transition cannot be taken until A is completed. (2) The transition aborts the action A. (3) The transition is taken and A is completed in the next mode. A scrutiny reveals that case (3) is unnecessary if the action is instead associated with a mode at a higher level. Intuitively, if the transition from M1 to M2 is taken when action A is in progress, the control information on the execution of action A will be lost. In other words, if action A can be in progress while the system is in M1 or M2, then action A should be associated with a mode parallel to M1 and M2. Of course, appropriate conditions must be specified in the new mode to ensure that the action is invoked only when the system enters mode M1.

Cases (1) and (2), however, must be formally captured in the formulas expressing mode transitions and the timing constraints imposed on actions. The remainder of this section discusses the modeling of sporadic and periodic timing constraints in Modechart. In each case, we give the RTL formulas capturing the semantics of mode transitions and timing constraints.

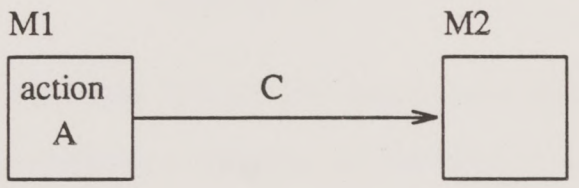


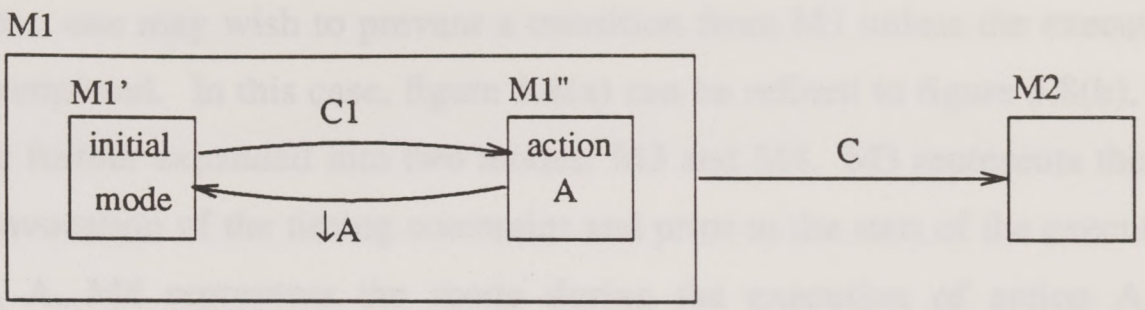
Figure 5.7

Sporadic Timing Constraint

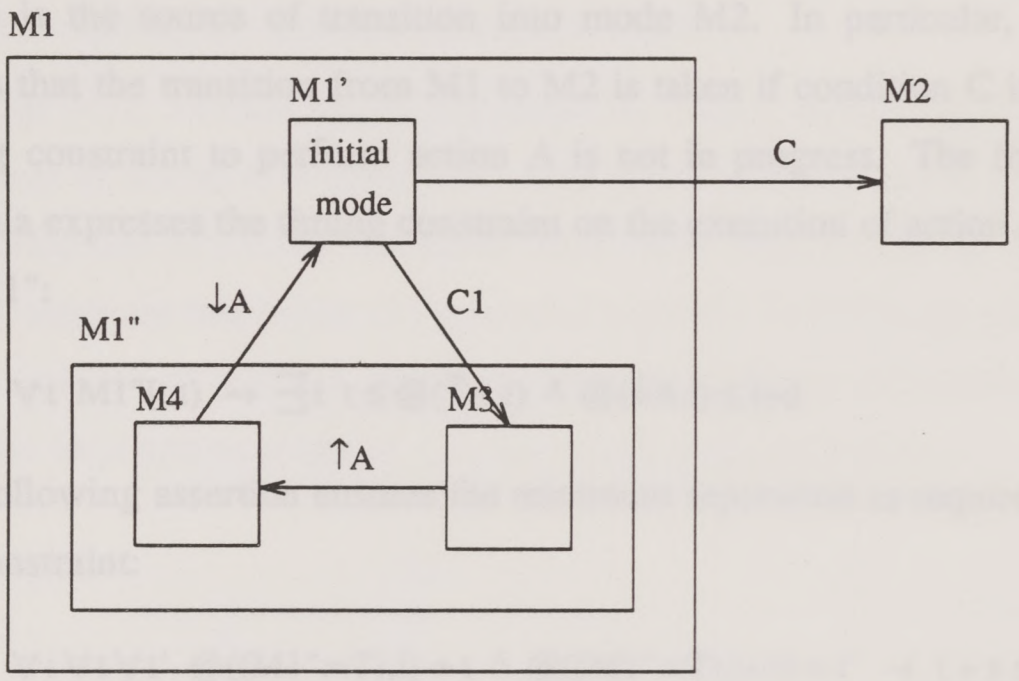
The syntax for a sporadic timing constraint in a mode M1 is:

When in mode M1, if condition C1 is true,
execute action A with deadline=d, separation=s.

If condition C1 becomes true while the system is in mode M1 or if C1 is true when the system enters mode M1, the timing constraint is invoked requiring action A to be performed before the deadline d. We say that the timing constraint is *in effect* from the time of its invocation (when in M1 and C1 holds) to the time when action A is completed (before the deadline d). The *separation* parameter *s* is the minimum time that must pass between two successive invocations of the sporadic timing constraint. Figure 5.8(a) illustrates how a sporadic timing constraint can be modelled in Modechart. Mode M1 contains two modes M1' and M1". The system is initially in mode M1' upon entering M1. When condition C1 is true, the transition from M1' to M1" is taken. The timing constraint is in effect while the system is in mode M1". Upon entering mode M1", action A is performed with the deadline d. When action A is completed, the transition is taken back to the initial mode M1'. The separation parameter specifies the minimum time that must elapse between two successive entries to mode M1".



(a)



(b)

FIGURE 5.8

The transition arrow from M1 to M2 represents the transition out of mode M1. The label C on the transition denotes an arbitrary condition specified by the system designer for exiting mode M1. We now consider the effect of a mode transition on the action A while the timing constraint on A is in effect.

a) Transition upon completion of action A: If the timing constraint is in

progress, one may wish to prevent a transition from M1 unless the execution of A is completed. In this case, figure 5.8(a) can be refined to figure 5.8(b). where M1" is further expanded into two modes: M3 and M4. M3 represents the mode after invocation of the timing constraint and prior to the start of the execution of action A. M4 represents the mode during the execution of action A. The motivation for exanding M1" is to illustrate that one can model a more detailed execution of action A by identifying the control information denoted by modes M3 and M4. However, the most important change from figure 5.8(a) to figure 5.8(b) is the source of transition into mode M2. In particular, figure 5.8(b) shows that the transition from M1 to M2 is taken if condition C is true and the timing constraint to perform action A is not in progress. The following RTL formula expresses the timing constraint on the execution of action A upon entering M1":

$$\forall t\ M1"[t,t) \rightarrow \exists i\ t \leq @(\uparrow A,i) \wedge @(\downarrow A,i) \leq t+d$$

The following assertion ensures the minimum separation as required by the timing constraint:

$$\forall i \forall t \forall t'\ @((M1":=T),i) = t \wedge @((M1":=T),i+1) = t' \rightarrow t + s \leq t'$$

The mode transition assertion for the transition from M1 to M2 follows from the discussion in section 5.3.2. In particular, if the condition on the transition arrow is a triggering condition, the following RTL formula captures the mode transition:

$$\forall t\ M1'(t,t) \wedge C \rightarrow \exists j\ @((M1'\text{-}M2),j) = t$$

where C in the above formula is the corresponding RTL predicates specifying the transition condition. However, if the condition on the transition arrow is a lower/upper bound restriction, the following RTL formula is applicable:

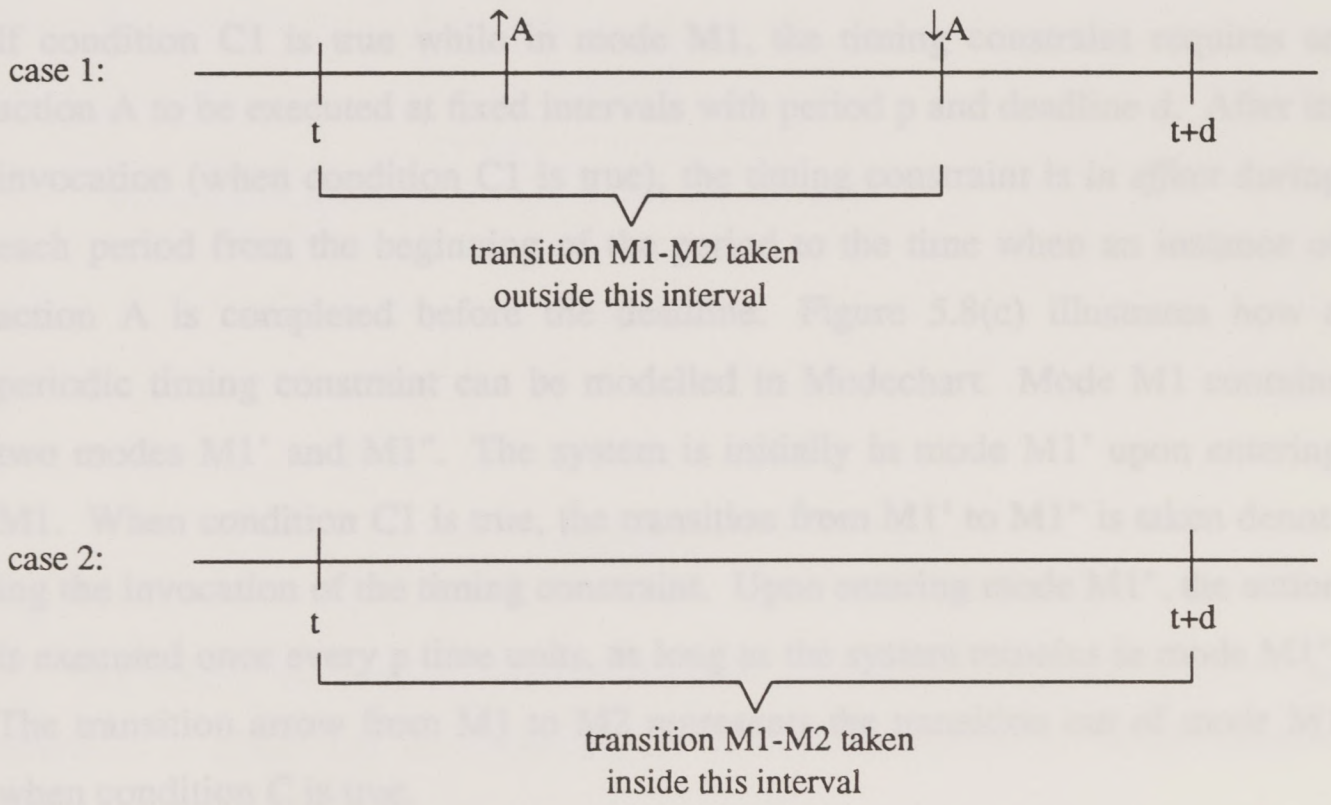
$$\forall t_1,t_2\ M1[t_1,t_2) \wedge M1'(t_2,t_2) \wedge t_1 + r \leq t_2 \rightarrow$$

$$\exists t_3 \exists \mathrm{j}\ @((\mathrm{M1'\text{-}M2}),\mathrm{j}) = t_3 \wedge t_1 + \mathrm{r} \leq t_3 \leq t_1 + \mathrm{d}$$

where r and d are the lower and upper bounds specified on the transition.

b) Transition aborting action A: One may wish to terminate action A or not start it when a transition to a new mode occurs. In figure 5.8(a), if condition C on the transition arrow becomes true while the system is in mode M1, the transition from M1 to M2 is taken even if the timing constraint is in effect. Hence, the mode transition assertion is very similar to case (a), except that the formula must now specify the transition from M1 to M2.

The RTL formula which expresses the timing constraint imposed on the execution of action A in mode M1" must now reflect the fact that action A may be aborted due to a mode transition from M1 to M2. As shown in the diagram below, there are two cases: (1) transition from M1 to M2 is not taken until after the completion of action A, or (2) transition from M1 to M2 is taken prior to the deadline imposed on action A, i.e., the action is not performed.



The following formula expresses the timing constraint on the execution of

action A which may be aborted. We assume that action A is atomic. Hence, if it is aborted due to a transition from M1, it is up to the implementation to ensure that no inconsistencies exit in the system state. The two cases mentioned above are captured by the two disjuncts in the formula:

$$
\begin{aligned}
&\forall t\ M1"[t,t) \rightarrow \\
&\quad (\exists\!\!\!-\, i\ \ t \le @(\uparrow A,i)\ \wedge\ @(\downarrow A,i) \le t+d\ \wedge \\
&\quad (\forall j\ @((M1\text{-}M2),j) < t\ \vee\ @(\downarrow A,i) < @((M1\text{-}M2),j))) \\
&\quad \vee \\
&\quad (\exists\!\!\!-\, j\ \ t \le @((M1\text{-}M2),j) \le t+d\ \wedge \\
&\quad (\forall i\ @(\downarrow A,i) \le t\ \wedge\ @((M1\text{-}M2),j) \le @(\uparrow A,i)))
\end{aligned}
$$

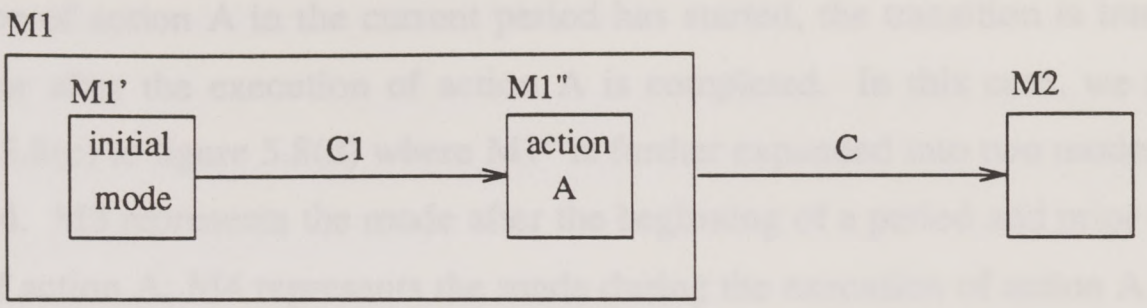
Periodic Timing Constraint

A periodic timing constraint in a mode M1 is of the form:

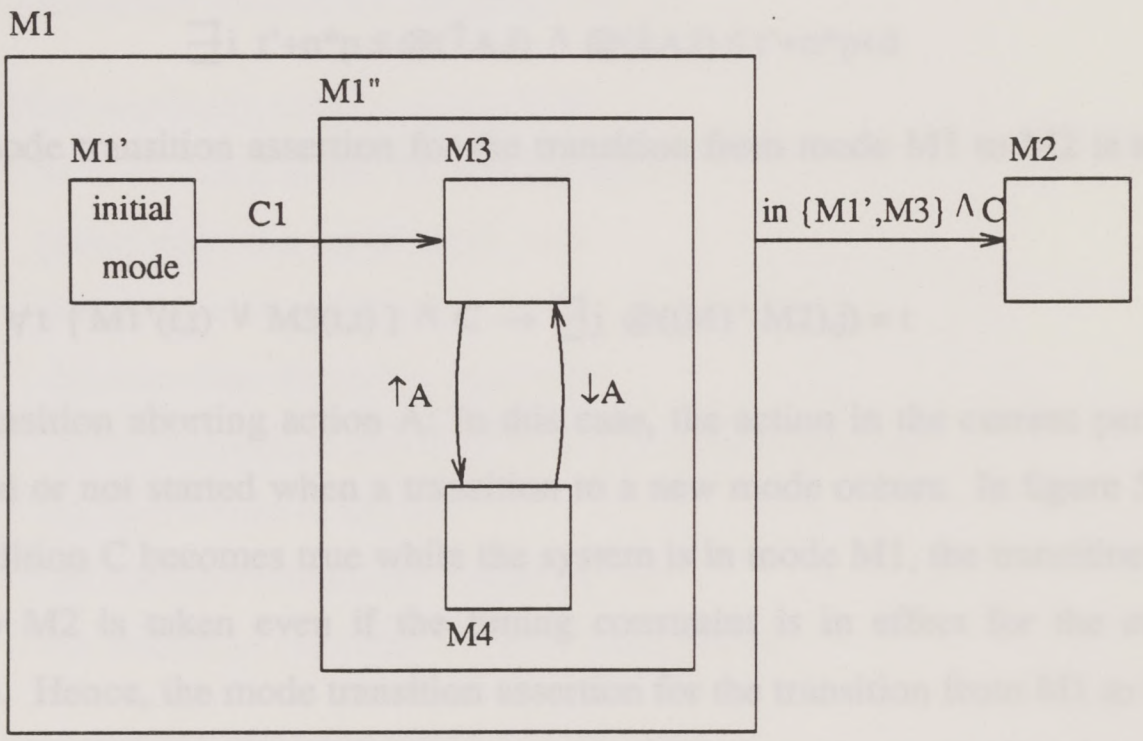
While in M1, if condition C1 is true,

execute A with period = p and deadline = d.

If condition C1 is true while in mode M1, the timing constraint requires an action A to be executed at fixed intervals with period p and deadline d. After its invocation (when condition C1 is true), the timing constraint is *in effect* during each period from the beginning of the period to the time when an instance of action A is completed before the deadline. Figure 5.8(c) illustrates how a periodic timing constraint can be modelled in Modechart. Mode M1 contains two modes M1' and M1". The system is initially in mode M1' upon entering M1. When condition C1 is true, the transition from M1' to M1" is taken denoting the invocation of the timing constraint. Upon entering mode M1", the action is executed once every p time units, as long as the system remains in mode M1". The transition arrow from M1 to M2 represents the transition out of mode M1 when condition C is true.



(c)



(d)

FIGURE 5.8

As before, we consider the two possible cases where a periodic timing constraint is in effect and the condition for taking a transition out of the mode is true.

a) Transition upon completion of the action in the current period: if the

instance of action A in the current period has started, the transition is intended to occur after the execution of action A is completed. In this case, we refine figure 5.8(c) to figure 5.8(d) where M1" is further expanded into two modes: M3 and M4. M3 represents the mode after the beginning of a period and prior to the start of action A; M4 represents the mode during the execution of action A. The following RTL formula captures the timing constraint imposed on the execution of action A upon entering mode M1":

$$\forall t \forall t' \forall n \; M1"[t',t) \wedge n*p \leq t-t' \rightarrow$$
$$\exists i \; t'+n*p \leq @(\uparrow A,i) \wedge @(\downarrow A,i) \leq t'+n*p+d$$

The mode transition assertion for the transition from mode M1 to M2 is as follows:

$$\forall t \; [\, M1'(t,t) \vee M3(t,t) \,] \wedge C \rightarrow \exists j \; @((M1'\text{-}M2),j) = t$$

b) Transition aborting action A: In this case, the action in the current period is aborted or not started when a transition to a new mode occurs. In figure 5.8(c), if condition C becomes true while the system is in mode M1, the transition from M1 to M2 is taken even if the timing constraint is in effect for the current period. Hence, the mode transition assertion for the transition from M1 to M2 is as described in section 5.3.2. However, the RTL formula expressing the timing constraint imposed on the execution of action A in mode M1" must now reflect the fact that action A may be aborted due to a a mode transition from M1 to M2.

$$\forall t \forall t' \forall n \geq 0 \; M1"[t',t) \wedge n*p < t-t' \rightarrow$$
$$(\exists i \; t'+n*p \leq @(\uparrow A,i) \wedge @(\downarrow A,i) \leq t'+n*p+d \wedge$$
$$(\forall j \; @((M1\text{-}M2),j) < t' \vee @(\downarrow A,i) \leq @((M1\text{-}M2),j)))$$
$$\vee$$
$$(\exists j \; t'+n*p < @((M1\text{-}M2),j) \leq t'+n*p+d \wedge$$
$$(\forall i \; @(\downarrow A,i) \leq t'+n*p \vee @((M1\text{-}M2),j) \leq @(\uparrow A,i)))$$

The motivation of this formula is similar to that of the sporadic case.

5.5. Entering and Existing Nested modes

We now give RTL formulas to capture the meaning of transitions out of modes which are nested inside other modes. Firstly, entering and exiting nested modes must be precisely defined:

Definition: (explicit and implicit entry)
A transition *explicitly* enters a mode M if
- the arrow ends at M, or
- the arrow crosses the boundary into M.

A transition *implicitly* enters a mode M if
Case 1: M is an initial mode, M' the immediate parent of M is a serial mode, and
- the arrow ends at M', or
- the transition implicitly enters M'.

Case 2: M' the immediate parent of M is a parallel mode, and
- the arrow ends at M', or
- the transition implicitly enters M', or
- the transition explicitly enters a sibling of M.

Definition: (explicit and implicit exit)
A transition *explicitly* exits a mode M if
- the arrow originates from M, or
- the arrow crosses the boundary out of M.

A transition *implicitly* exits a mode M if
Case 1: M' the immediate parent of M is a serial mode, the system is in mode M, and
- the arrow originates from M', or
- the transition implicitly exits M'.

Case 2: M' the immediate parent of M is a parallel mode and

- the arrow originates from M', or
- the transition implicitly exits M', or
- the transition explicitly exits a sibling of M.

For example, in figure 5.9, the transition from M4 to M3 has explicit exits from modes M4 and M1, an explicit entry into M3, and implicit entries into modes M6 and M7. The transition from M3 to M1 has explicit exits from M3, implicit exits from modes M6 and M7, an explicit entry into M1, and an implicit entry into M4.

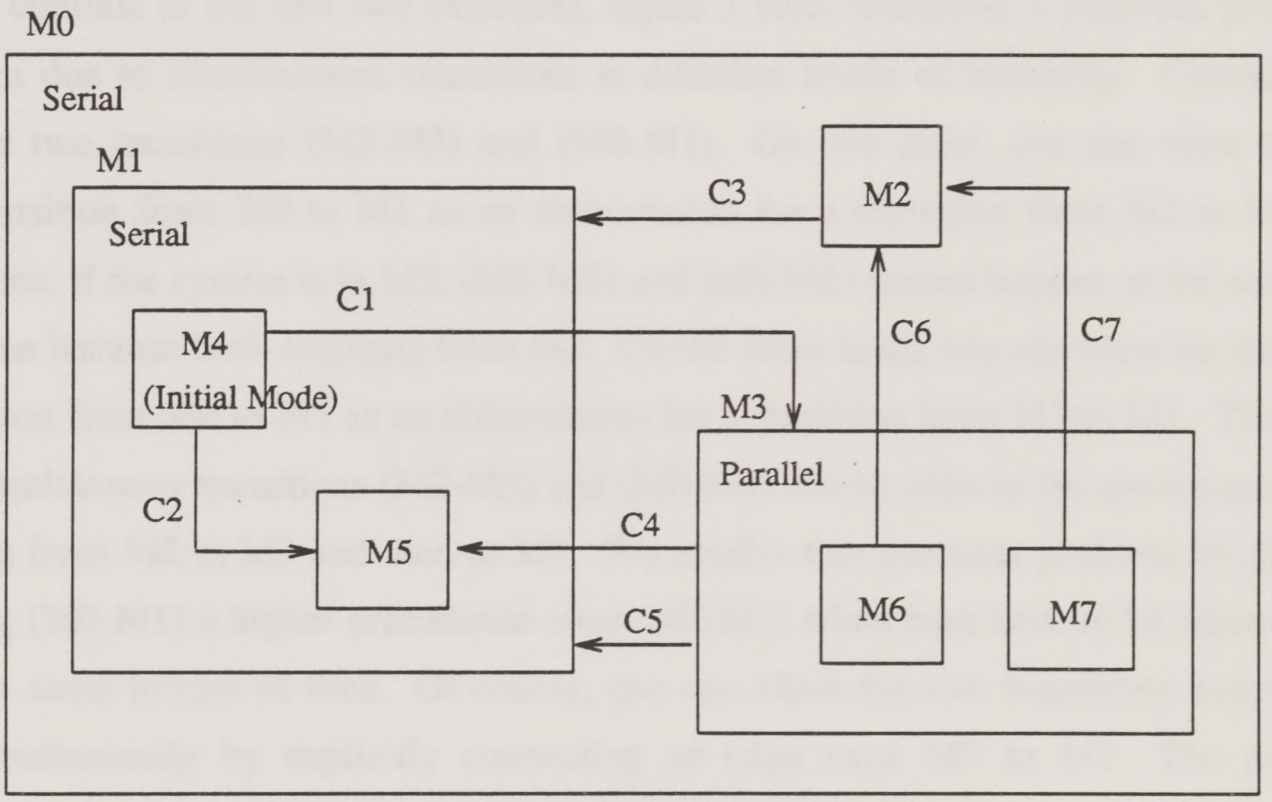


FIGURE 5.9

Definition: (*at level*)
Suppose mode M' is an immediate child of mode M, then mode M' is said to be *at level* M.

In figure 5.9, mode M1, M2, and M3 are at level M0, and M4 and M5 are at level M1. Modes M6 and M7 are at level M3.

Secondly, since a Modechart system can be in more than one mode simultaneously and multiple transitions can originate from a mode, it is possible that more than one transition can be taken at an instant of time. We identify a set of rules which identify simultaneous transitions that violate the semantics of Modechart. We motivate the discussion by considering three simple examples shown in figure 5.10. In figure 5.10(a), two transitions (M1-M2) and (M1-M3) originate from the same mode, thus both transitions cannot be taken at the same instant of time. However, in figure 5.10(b), a transition inside M1 can happen simultaneouysly with a transition inside M2, because M1 and M2 are in parallel. In contrast to the first two examples, figure 5.10(c) illustrates a potential problem due to simultaneous transitions at different levels of hierarchy. Consider the two transitions (M2-M3) and (M0-M1). On one hand, one can view the transition from M0 to M1 as an abbreviation for a transition from M2 to M1. Then, if the system is in M2, (M2-M3) and (M0-M1) cannot happen at the same time because both originate from M2. On the other hand, one can view the transition from M0 to M1 as an abbreviation for a transition from M3 to M1. Then, simultaneous transitions (M2-M3) and (M0-M1) can be seen as the system moving from M2 to M3 and then to M1. We resolve this potential problem by giving (M0-M1) a higher precedence over (M2-M3) when *both* have to be taken at the same instant of time. Of course, one can allow the two transitions happen simultaneouly by explicitly connecting an edge from M3 to M1. The precedence among transitions (explicitly or implicitly) exiting a mode is defined below:

(1) Transitions originating from the same mode have the same precedence.

(2) Transitions originating from modes in parallel and at the same level have the same precedence.

(3) Transitions originating from inside two distinct modes in parallel at level M and exiting out of M have the same precedence.

(4) Transitions originating from a mode M have precedence over transitions

originating from the children of M.

(5) Transitions originating from a mode M have precedence over transitions originating from children of a mode M', where M and M' are in parallel.

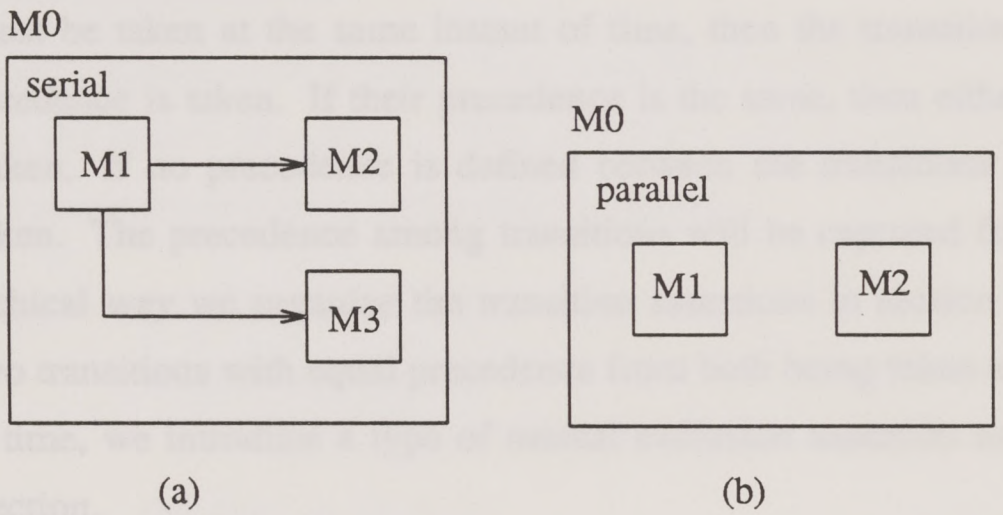


FIGURE 5.10

Referring to figure 5.9, (M4-M3) has the same precedence as (M4-M5) because of precedence rule (1). By rule (2), both (M6-M2) and (M7-M2) have the same precedence. (M3-M1) has precedence over both (M6-M2) and (M7-M2) by rule (4). Any transitions coming out from inside M6 and M7 would have had the same precedence by rule (3). Lastly, by rule (5), (M6-M2) would have had precedence over transitions coming from inside M7 and (M7-M2) would

have had precedence over transitions coming from inside M6.

Having defined the notion of precedence among transitions from a mode, we can now specify rules for simultaneous transitions. If two transitions out of a mode must be taken at the same instant of time, then the transition with the higher precedence is taken. If their precedence is the same, then either (but not both) is taken. If no precedence is defined between the transitions then both may be taken. The precedence among transitions will be captured formally by the hierarchical way we organize the transition assertions in section 5.5.3. To prevent two transitions with equal precedence from both being taken at the same instant of time, we introduce a type of mutual exclusion assertion as shown in the next section.

5.5.1. Mode transition exclusion

Let e_1 and e_2 be mode transition events corresponding to two distinct transitions with equal precedence, the exclusion assertion is:

$$\forall i \forall j \ @(e_1,i) \neq @(e_2,j)$$

Two transitions may have the same precedence if they fall under rules (1), (2) and (3) in the previous section. For each pair of transitions which falls under one of the three rules, an exclusion assertion is specified.

5.5.2. Implicit mode exits

When a transition is taken, a set of modes may be explicitly and implicitly exited and another set of modes explicitly and implicitly entered. As will be shown in section 5.5.3, a formula capturing a mode transition will specify explicit and implicit mode entries and explicit mode exits. The reason for including the explicit entries and implicit entries as part of a single mode transition assertion (section 5.5.3) is that they cannot be specified independent of the specific

transition taken to enter a compound mode. For example, in figure 5.9, entering mode M1 could mean entering M4 or M5 depending on the transitions. For implicit mode exits, however, the corresponding RTL formula can be written independent of the transitions.

Suppose a system exits at time t from a mode M which has as its immediate children the modes M_1, M_2, ... , M_n. If M is a serial mode, the following assertion captures the exit from its immediate children.

$$\begin{aligned} M(t,t] \rightarrow\ & (M_1(t,t) \rightarrow M_1(t,t]) \wedge \\ & (M_2(t,t) \rightarrow M_2(t,t]) \wedge \\ & \quad \cdots \\ & (M_n(t,t) \rightarrow M_n(t,t]) \end{aligned}$$

If M is a parallel mode, the assertion to capture implicit exits is simply:

$$M(t,t] \rightarrow M_1(t,t] \wedge M_2(t,t] \cdots \wedge M_n(t,t]$$

Finally, specifying explicit exits as part of the mode transition assertion allows us to use the above assertion for exiting all modes which are exited implicitly due to the transition.

5.5.3. The transition assertion for nested modes

We shall use a single assertion to capture the transitions in a set of nested modes. The mode transition assertion is structured to reflect the serial and parallel structure of the system. Intuitively, a formula *at a level* M describes the system behavior specified by the modes nested in M. At a level M, we include all the formulas describing the transitions originating from each immediate child M' of M. In turn, the level M' formulas describe the modes inside M', their transitions and timing constraints. (A complete specification will also include the formulas capturing the timing constraints imposed on the actions in M'. For the purpose of this section, we shall not be concerned with timing constraints.

To deal with sporadic and periodic timing constraints, the transition assertion given here can be modified in a straightforward manner in accordance with the discussion in section 5.4.2.)

Suppose M is a serial mode and M' is an immediate child of M. Let n be the number of transitions from M' with a triggering condition on the arrow and let $C_1, C_2, \cdots, C_n$ be the corresponding triggering conditions. Let m be the number of transitions from M' with a lower/upper bound condition on the arrow and let $B_1, B_2, \cdots, B_m$ be the corresponding bounds on the transitions. (The C_is and B_is are RTL predicates as described in section 5.3.2.) The following formula captures the transitions originating from an immediate child M' of mode M:

{Level M transition assertion is the conjunction of formulas below for each M'}

$$(M'(t,t) \wedge C_1 \rightarrow T_1) \vee (M'(t,t) \wedge C_2 \rightarrow T_2) \vee \cdots \vee (M'(t,t) \wedge C_n \rightarrow T_n) \quad (1)$$

$$\begin{aligned} M'[t,t) \rightarrow\ & [\,(T_{n+1} \wedge B_1) \vee (M'[t,t'] \wedge t' \leq t + d_1)\,] \wedge \\ & [\,(T_{n+2} \wedge B_2) \vee (M'[t,t'] \wedge t' \leq t + d_2)\,] \wedge \\ & \cdots \\ & [\,(T_{n+m} \wedge B_m) \vee (M'[t,t'] \wedge t' \leq t + d_m)\,] \end{aligned} \quad (2)$$

$$\begin{aligned} & T_1 \rightarrow \Theta_1 \\ & T_2 \rightarrow \Theta_2 \\ & \cdots \\ & T_{n+m} \rightarrow \Theta_{n+m} \\ & M'(t,t) \wedge \neg (T_1 \vee T_2 \vee \cdots \vee T_{m+n}) \rightarrow \{\text{Level M' formulas}\} \end{aligned} \quad (3)$$

where T_i is a predicate denoting the occurrence of the mode transition event corresponding to the *i*th transition from M', and each Θ_i is the conjunction of

- the mode predicates denoting explicit exits for the *i*th transition, and
- the mode predicates denoting explicit and implicit mode entries for the *i*th transition

Formula (1) specifies the transitions from M' with triggering conditions. Formula (2) specifies the transitions from M' with lower/upper bounds. Formula (3) specifies the condition for level M' formulas by asserting that if the system is in M' and none of the transitions originating from M' has been taken at time t, then the assertions about the mode transitions nested in M' will hold.

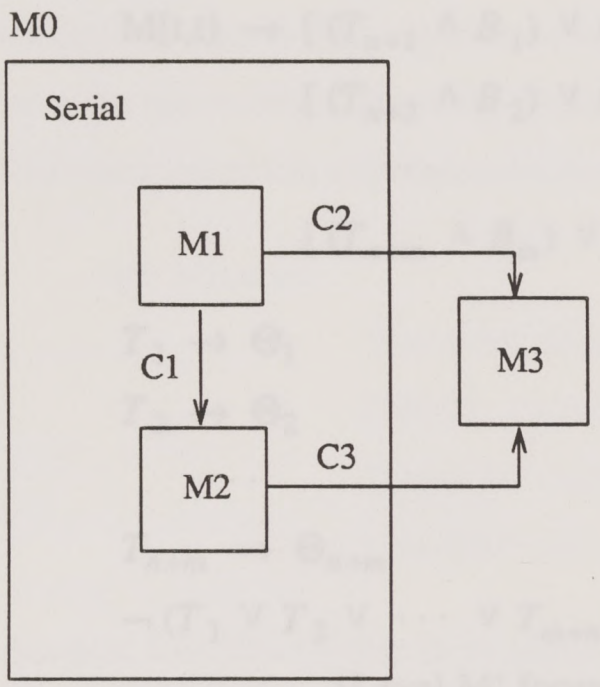


FIGURE 5.11

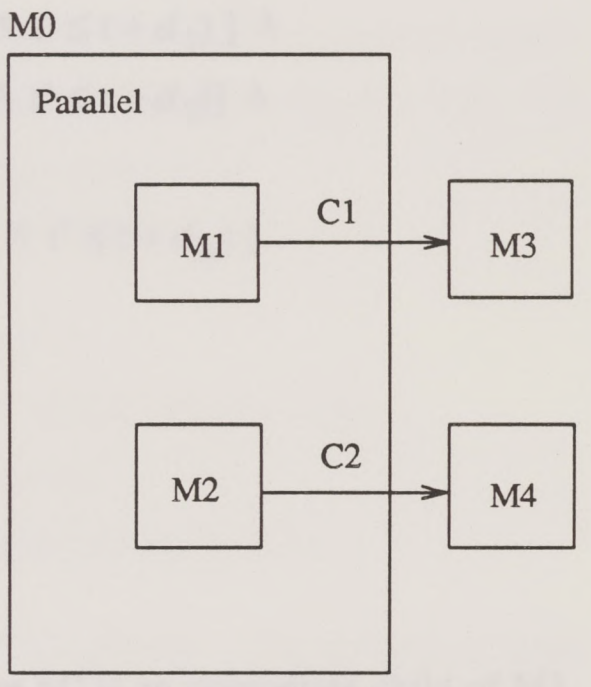


FIGURE 5.12

For example, assume the conditions on the transitions in figure 5.11 are all triggering conditions. The level M0 mode transition assertion is:

{Level M0 transition assertion}

$(M1(t,t) \wedge C1 \rightarrow \exists i\ @((M1\text{-}M2),i) = t) \vee$

$(M1(t,t) \wedge C2 \rightarrow \exists i\ @((M1\text{-}M3),i) = t)$

$\forall i\ @((M1\text{-}M3),i) = t \rightarrow M1(t,t] \wedge M0(t,t] \wedge M3[t,t)$

$\forall i\ @((M1\text{-}M2),i) = t \rightarrow M1(t,t] \wedge M2[t,t)$

$\forall i,j\ M1(t,t) \wedge @((M1\text{-}M3),i) \neq t \wedge @((M1\text{-}M2),j) \neq t \rightarrow$ {Level M1}

$M2(t,t) \wedge C3 \rightarrow \exists i\ @((M2\text{-}M3),i) = t$

$\forall i\ @((M2\text{-}M3),i) = t \rightarrow M2(t,t] \wedge M0(t,t] \wedge M3[t,t)$

$\forall$i M2(t,t) $\wedge$ @((M2-M3),i) $\neq$ t $\rightarrow$ {Level M2 formulas}

If M is a parallel mode, the following formula captures the transitions originating from all immediate children of mode M:

{Level M transition assertion is the conjunction of the following formulas}

$(C_1 \rightarrow T_1) \vee (C_2 \rightarrow T_2) \vee \cdots \vee (C_n \rightarrow T_n)$

$M[t,t) \rightarrow [\,(T_{n+1} \wedge B_1) \vee (M[t,t'] \wedge t' \leq t + d_1)\,] \wedge$

$[\,(T_{n+2} \wedge B_2) \vee (M[t,t'] \wedge t' \leq t + d_2)\,] \wedge$

$\cdots$

$[\,(T_{n+m} \wedge B_m) \vee (M[t,t'] \wedge t' \leq t + d_m)\,]$

$T_1 \rightarrow \Theta_1$

$T_2 \rightarrow \Theta_2$

$\cdots$

$T_{n+m} \rightarrow \Theta_{n+m}$

$\neg (T_1 \vee T_2 \vee \cdots \vee T_{m+n}) \rightarrow$

{Level M' formulas where M' is an immediate child of M}

where C_i, B_i, T_i, Θ_i, and d_i are as described in the serial case earlier. Observe that the above formulas holds when the the system is in mode M at time t. Since M is a parallel mode, it is not necessary to specify the mode M' in the antecedents of the implications of the above formulas.

As an example, assume both conditions on the transitions in figure 5.12 are triggering conditions. The level M0 mode transition assertion is as follows:

{Level M0 transition assertion}

(C1 $\rightarrow \exists$i @((M1-M3),i) = t) $\vee$

(C2 $\rightarrow \exists$i @((M2-M4),i) = t)

$\forall$i @((M1-M3),i) = t $\rightarrow$ M1(t,t] $\wedge$ M0(t,t] $\wedge$ M3[t,t)

$\forall$i @((M2-M4),i) = t $\rightarrow$ M2(t,t] $\wedge$ M0(t,t] $\wedge$ M4[t,t)

$\forall$i $\forall$j @((M1-M3),i) $\neq$ t $\wedge$ @((M2-M4),j) $\neq$ t $\rightarrow$ {Levels M1 and M2}

We conclude this section by discussing how the semantics of Modechart deals with the potential ambiguities caused by simultaneous transitions. The precedence rules and the transition exclusion assertions handle the case where two transitions can (explicitly or implicitly) exit a mode. If the two transitions have the same precedence, either one (but not both) can be taken. Otherwise, the transition with higher precedence is taken. However, the semantics of Modechart allows that any two concurrent transitions inside two modes in parallel to be taken simultaneously. Recall the parallel mode M in figure 5.2(b). The following assertion expresses the two mode transitions in M.

$$\forall t \; M(t,t) \rightarrow$$

{Level M' transition assertion}

$$[\, M1(t,t) \wedge M3(t,t) \rightarrow \exists i \; @((M1\text{-}M2),i) = t \,] \wedge$$

{Level M" transition assertion}

$$[\, M3(t,t) \wedge M1(t,t) \rightarrow \exists i \; @((M3\text{-}M4),i) = t \,]$$

Assume that the system is in mode M1 and M3 initially. The preceding assertion allows the two transitions, from M1 to M2 and from M3 to M4, to occur at the same time. If there are lower/upper bounds on the transitions, then the semantics of Modechart does not require both transitions to be taken.

5.6. Absence of Anomalous Behavior

In the preceding sections, we described the syntax and provided a formal semantics for Modechart in terms of Real Time Logic. In this section, we give a theorem which shows that the well-formedness requirement and the UDIM condition ensure the proper behavior of serial and parallel modes as described in section 5.2. We also state a condition and give a theorem for preventing the anomaly which arise from cycles of transitions.

Lemma 5.1:

If the root mode of a system is well-formed and the UDIM condition holds, then

(a) if a transition (explicitly or implicitly) enters a serial mode M, it enters exactly one of the immediate children of M.

(b) if a transition (explicitly or implicitly) enters a parallel mode M, it enters all of the immediate children of M.

(c) if a transition exits from a (serial or parallel) mode M, it exits from all of the immediate children of M.

Proof: The proof follows directly from the well-formedness assumption, the UDIM condition, and the precedence rules described in the previous section. □

Theorem 5.2:

Suppose the root mode of a system is well-formed, and the UDIM condition holds for the system. Suppose M is a compound mode and M_i, M_j are two of its immediate children.

(a) If M is a serial mode, then if the system is in mode M_i for a non-zero interval of time, it cannot be in another mode M_j for non-zero units of time in the same interval.

(b) If M is a parallel mode, then if the system is in mode M_i for a non-zero interval of time, it is also in each mode M_j during the same time interval.

Proof: We prove this theorem by induction. We start with the top-most mode as the base case. We then show that if properties (a) and (b) hold for all ancestors of a mode, they also hold for the mode.

Base Case: (top-most mode)

(A) Consider the case where the top-mode mode M is a serial mode. Let M_i and M_j be two immediate children of M. Lemma 1 implies that the system will be in exactly one mode upon system start-up.

Assume that the system is in mode M_i during the interval from t_i to t'_i where $t_i < t'_i$. If the system enters another mode M_j at time t_j where $t_i \leq t_j < t'_i$, it cannot remain in mode M_j for any non-zero time interval. Otherwise, the system must have exited M_i at time t_j and stayed out of M_i for a non-zero time interval.

(B) Consider the case where the top-most mode M is a parallel mode. Let M_i and M_j be two immediate children of M. We know by lemma 1 that the system starts in both M_i and M_j. Furhermore, the well-formedness property implies that there is no transition connecting the two modes M_i and M_j. Hence, the system will remain in both modes.

Induction Step:

(A) Let M be a serial mode such that properties (a) and (b) in the statement of the theorem hold for ancestors of M. We need to show that property (a) holds for M. Again, let M_i and M_j be two immediate children of M.

Assume the system is in mode M_i during the interval from t_i to t'_i where $t_i < t'_i$. Furthermore, assume the system enters another mode M_j at time t_j where $t_i \leq t_j < t'_i$. The system cannot remain in mode M_j for any non-zero time interval for the following reason.

M_j is entered because of the occurrence of a sequence of transitions at time t_j. The first mode in this sequence of transitions can be either one of the two cases:

(i) The sequence of transitions originated from mode M_i. But this is impossible, because it means that the system is not in M_i from t_j to t'_j+1.

(ii) The sequence of transitions originated from a mode M' outside mode M. From the well-formedness assumption, we know that M (or its ancestor) is in series with M' (or its ancestors). This fact implies

that the system was simulataneously in M and M' right before time t_j for a non-zero interval of time. This is a contradiction to the hypothesis of the induction step.

(B) Let M be a parallel mode such that properties (a) and (b) in the statement of the theorem hold for ancestors of M. We need to show that property (b) holds for mode M as well.

Assume the system is in mode M_i during the interval from t_i to t'_i where $t_i < t'_i$. Furthermore, assume that the system is in mode M_j until it exist M_j at time t_j where $t_i \leq t_j < t'_i$. If the system remains outside M_j in another mode M' for some non-zero time interval, then by the well-formedness assumption we know that M (or its ancestor) is in series with M' (or its ancestor. But this is a contradiction to the hypothesis of the induction step, because it implies that the system is in two modes (in series) simultaneously during the interval from t_j to t_j+1. □

In section 5.2, we described an anomaly which may result from a cycle of transitions. Clearly, it is too restrictive to disallow transition cycles in a specification. However, if we can ensure that there is a positive delay or an action is executed in one of the modes involved in a cycle of transitions, the anomaly is avoided. The following definitions will be used in a sufficient condition for preventing the transition cycle anomaly.

Definition: (cycle of transitions)

Let $T_1, T_2, \cdots, T_m$ denote a sequence of transitions. The transitions form a *cycle* on a set of modes $M_1, M_2, \cdots, M_m$ if

- each transition T_i (implicitly or explicitly) exits M_i and (implicitly or explicitly) enters M_{i+1} for all $1 \leq i < m$, and
- transition T_m exits mode M_m and enters M_1.

Definition: (Blocked Transition)

A transition exiting mode *M* and entering mode *M'* is *blocked* if
- there is an action associated with mode *M* such that the transition cannot be taken until that action completes, or
- there is a positive delay imposed on the transitions.

Theorem 5.3:

A Modechart specification is free from the transition cycle anomaly if in each cycle of transitions, at least one transition *t* exiting mode *M* and entering *M'* is blocked by *M*.

Proof: If a transition in a cycle is blocked by an action or a delay, the cycle cannot be traversed instantaneously. Hence, the anomaly is avoided. □

For each transition cycle, the preceding theorem ensures that the system remain in at least one mode for non-zero units of time. This is possible because either at least one action is executed in one of the modes involved in the cycle, or there is a positive delay on one of the transitions in the cycle. We remark that the above condition is not necessary to prevent the transition cycle anomaly.

5.7. Conclusion

In this chapter, we have presented a language called Modechart for the specification of real-time systems. Modechart owes its origin to the *mode* concept of Parnas *et al* in their systems requirement work on the A-7E aircraft and also to the Statechart language of Harel *et al*. The emphasis of Modechart, however, is in the specification of absolute timing properties. A formal semantics of Modechart was given in terms of RTL (Real Time Logic).

The concept of modes is familiar to designers of process control systems. Hence, Modechart should be an easy specification language to use for defining

real-time systems. More importantly, the specification of a system in terms of modes gives us more leverage in the verification of timing properties. Serial and parallel modes provide a way to organize the assertions about system behavior (RTL formulas) in a hierarchical and compartmental organization. Proof techniques can take advantage of this organization to focus on the set of relevant assertions to establish a timing property. The ability to avoid considering all the assertions defining the behavior of a large system is a prerequisite to practical applications of verification technology.

A preliminary version of a software tool which allows a user to describe a system specification in Modechart and a translator for generating the corresponding RTL formulas has been developed as part of the SARTOR project. The graphics-based tool for generating a Modechart specification, the Modechart Constructor consists of two components: (1) an icon-driven user interface for creating, displaying and modifying a specification, and (2) a database manager from which the information is retrieved by the user interface through message passing.

The software tool, TRANS for generating RTL formulas from a Modechart specification performs the translation in two stages. First, TRANS converts the Modechart specification in the database into an intermediate ASCII format. Then it generates the appropriate RTL formulas as queried by the user. The primary advantage of a two-stage translation is that it allows the user to bypass the Modechart Constructor tool when a bit-map workstation is not available. A system specification in the ASCII format can also be used directly as input to the TRANS software for generating the corresponding RTL formulas.

Chapter 6

Verification of Modechart Specifications

6.1. Introduction

The primary difficulty in deciding the satisfiability (or unsatisfiability) of an RTL formula is the potentially infinite number of event occurrences that must be considered. This problem arises due to the use of both arbitrary quantification and the uninterpreted functions in RTL formulas. The occurrence function does not pose a problem in obtaining decidability, because an occurrence function cannot be an argument to an uninterpreted function. However, skolemizing an existentially quantified variable may yield an uninterpreted function which may be an argument to other uninterpreted functions.

Even after using serial and parallel modes in specifying a system, the corresponding RTL formulas involve arbitrary $\forall$ and $\exists$ for variables denoting event occurrences. In order to take advantage of the hierarchical and orthogonal control information that is provided by a Modechart specification, we must avoid verifying a given property against the entire set of RTL formulas. The objective of this chapter is to examine how a system property specified in RTL can be verified with respect to a Modechart specification without explicitly examining all possible occurrences of events in the system.

6.2. The Approach

The objective is to use the Modechart specification to consider only a finite number of event occurrences to verify a system property. First, we define the computations of a system specified in Modechart as a directed tree augmented with a set of relations expressing the lower/upper bounds on pairs of events. If we can represent the potentially infinite computation tree by a finite graph, then

if a procedure can decide a class of properties for this graph, it is a decision procedure for the original Modechart specification.

Figure 6.1 and Figure 6.2 illustrate the proposed approach. A Modechart specification is used to generate a finite *computation graph*. While generating the graph, the corresponding RTL formulas can be used to prune the vertices that are unreachable due to the timing constraints. These RTL formulas pose little difficulty in an analysis because constants are used for event instances in the occurrence functions. The key is in generating the computation graph so that it accurately reflects the modechart specification. A model for a modechart specification is an assignment of times to events. Likewise, a model of a computation graph is an assignment of times to the events on a path in the graph. A model for a Modechart specification, i.e., for the RTL formulas that capture the formal semantics of the specification, must be shown to be a model for the computation graph. Then if we prove a property about the computation graph, it also holds for the original Modechart specification (Figure 6.1). However, if we can also show that a model for the computation graph is a model for the Modechart specification, then a decision procedure for a class of properties for the computation graph is also a decision procedure for the Modechart specification (Figure 6.2).

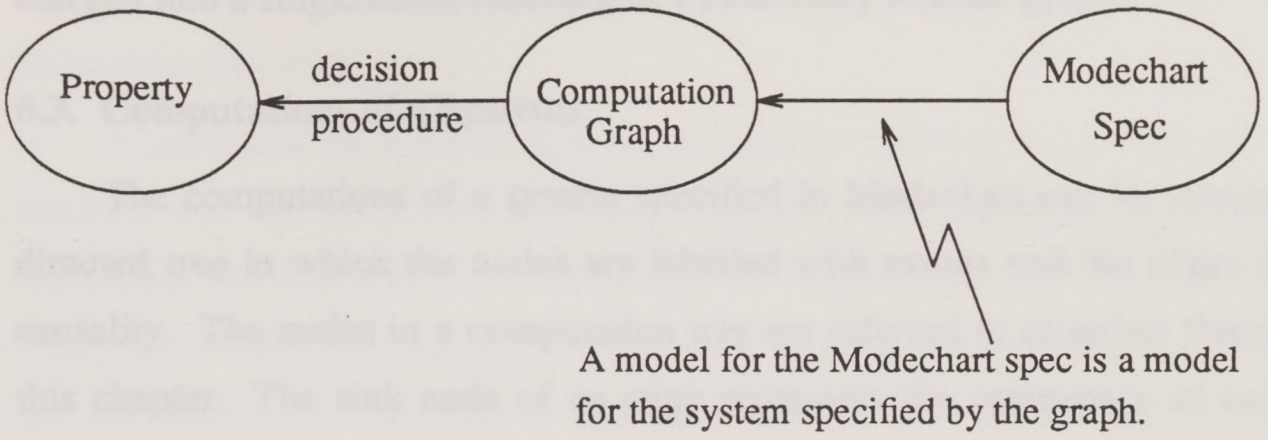


FIGURE 6.1

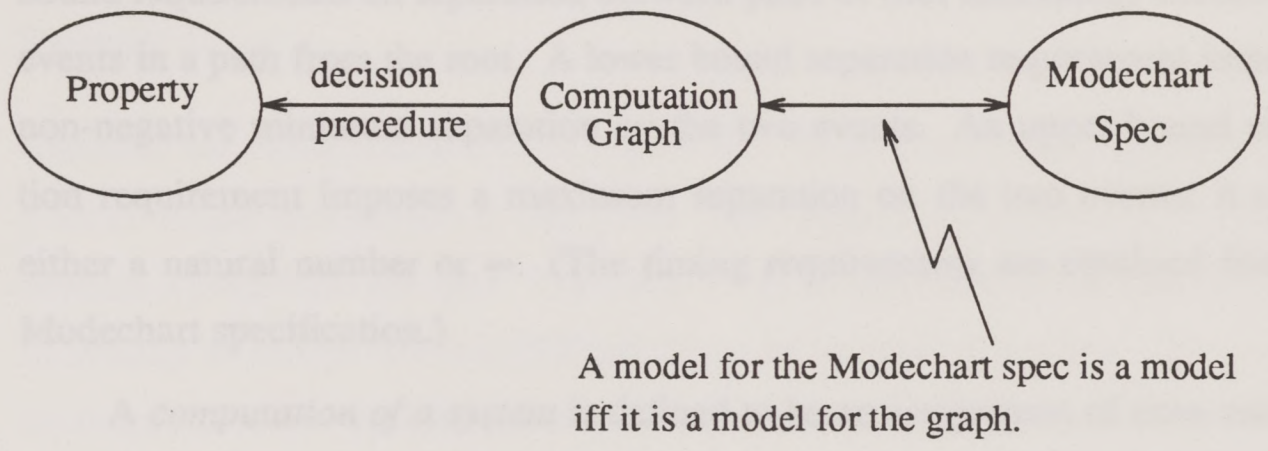


FIGURE 6.2

Although the approach proposed in this paper is similar to the reachability analysis for timed-petri nets, e.g.,[Razouk & Phelps 84], [Zuberek 1980], [Ramchandani 74], [Merlin & Farber 76], there are three major differences. First, the model of computation used in the petri net approach relies on interleaving. Second, the restrictions imposed on the types of nets (such as decision free nets, free choice nets and nets with conflict sets) and the restrictions on the timing constraints limit the systems that can be described. Lastly, the reachability algorithms, for example the one presented in[Razouk & Phelps 84], rely on the presence of counters in the nodes of the reachability graph to measure elpsed times. The approach in this paper allows nodes with different timer values to be

merged into a single node, resulting in a potentially smaller graph.

6.3. Computations of a Systems

The computations of a system specified in Modechart can be viewed as a directed tree in which the nodes are labelled with events and the edges denote causality. The nodes in a computation tree are referred to as *points* throughout this chapter. The sink node of an edge represents the occurrence of an event caused by the events on the path from the root to that point. A computation tree is augmented by a set of timing requirements representing the lower/upper bound requirements on separation between pairs of (not necessarily consecutive) events in a path from the root. A lower bound separation requirement imposes a non-negative minimum separation on the two events. An upper bound separation requirement imposes a maximum separation on the two events; it can be either a natural number or ∞. (The timing requirements are obtained from the Modechart specification.)

A *computation of a system* is defined to be an assignment of time values to the events on a path (perhaps infinite) from the root of the tree such that it is consistent with lower/upper bound requirements on the events. Consider a path consisting of a sequence of points $P_0, P_1, \ldots$. Each point P_i represents an event occurrence. An edge from P_i to P_{i+1} represents the occurrence of an event at point P_{i+1} which was caused by the events from P_0 to P_i. The timing requirements of the system, as described by the Modechart specification, impose minimum/maximum separation on the points (actually event occurrences) in the path. For a pair of points P_i and P_j on a path where i<j, the timing requirements can be represented as

$$P_i + \mathrm{I} \leq P_j \qquad \text{(lower bound separation requirement)}$$

$$P_j \leq P_i + \mathrm{I} \qquad \text{(upper bound separation requirement)}$$

where I is a non-negative integer. If no restriction is imposed on the two points,

it is assumed that the minimum separation from P_i to P_j is at least zero, and the maximum separation is ∞.

A point can be labelled with more than one event if the events are simultaneous. For example, a point labelled with the mode transition event (M1-M2) represents the simultaneous occurrence of two events, the mode exit event (M1:=F) and the mode entry event (M2:=T). However, two simultaneous events are not necessarily labelled at the same point. Two distinct points can be labelled each with one of the simultaneous events, but the lower/upper bound separations must then require the events to happen at the same time.

Example 1:

Consider the computation tree in Figure 6.3. Each point (node) is labelled with a set of events that happen at that point. The sequence of points $P_0, P_1, P_2, \cdots$ denotes a path from the root. The point P_0 is the root indicating that the system is initially in modes M1 and M3 and the state variable S is false. The point P_1 corresponds to the occurrence of the event (M1-M2) which denotes that a mode transition from M1 to M2 is taken at that point. The following lower/upper bound separations may be imposed on the points in the path segment P_0 to P_5. The inequalities are intended to denote bounds on the instances of events corresponding to each point.

$P_0 + 10 \leq P_1$	delay on transition M1 to M2
$P_2 + 20 \leq P_3$	computation time of action A
$P_1 + 150 \leq P_5$	delay on transition from M2 to M4
$P_3 \leq P_1 + 100$	deadline on action A
$P_4 \leq P_3$	(S:=T) happens at same time as $\downarrow$A

The lower and upper bound separation between P_3 and P_4 are zero, thus the two points (i.e., the corresponding events) are simultaneous. An assignment of time values to the event occurrences in the above path such that they do not violate

the lower/upper bound requirements is a computation of the system.

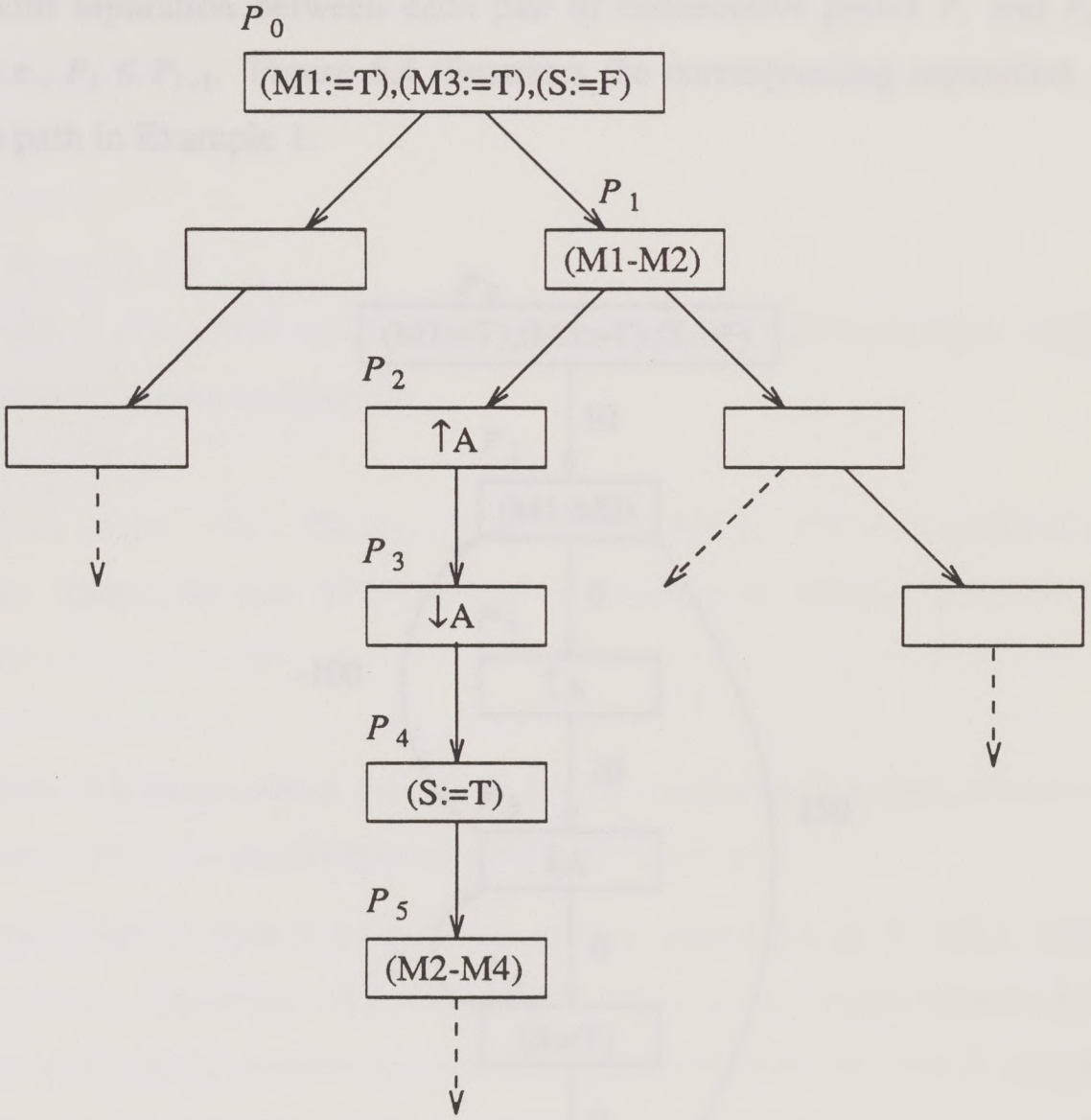


FIGURE 6.3

The set of bound separations on a path in a computation tree can in turn be represented by a directed graph, referred to as a *separation graph.* The nodes in this graph are the points in the corresponding path and the weights on the edges denote the specified separation between the nodes. Specifically, given a pair of points P_i and P_j on a path, a separation between the points,

$$P_i + \mathrm{I} \leq P_j$$

is represented in the graph by an edge from P_i to P_j with the weight I. A separation graph is connected, because by definition there is at least zero minimum separation between each pair of consecutive points P_i and P_{i+1}in a path, i.e., $P_i \leq P_{i+1}$. Figure 6.4 illustrates the corresponding separation graph for the path in Example 1.

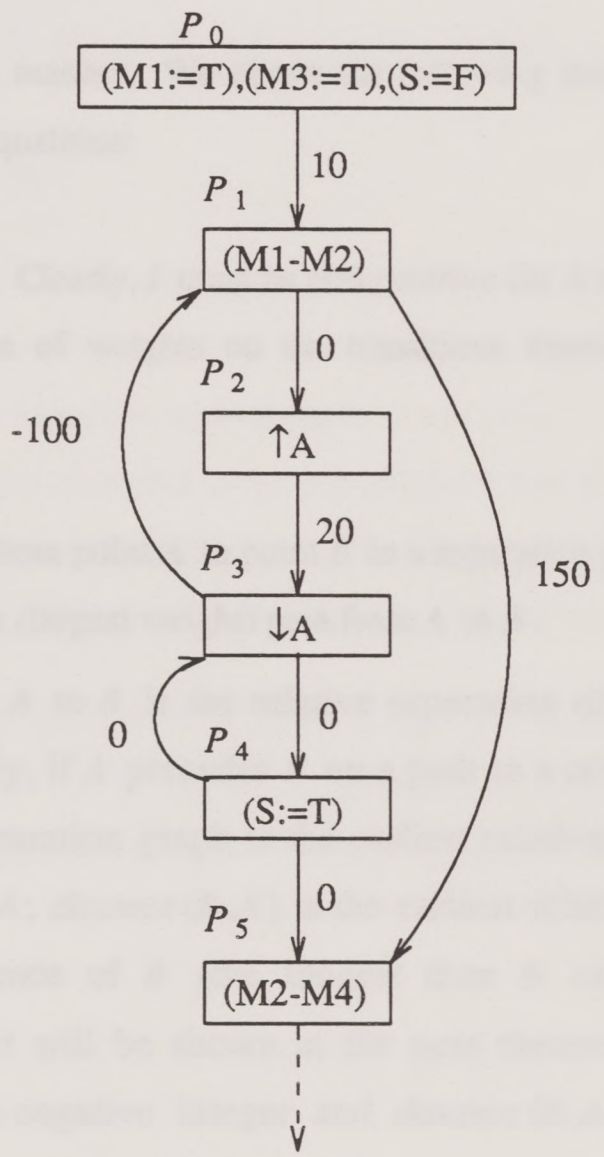


FIGURE 6.4

Theorem 6.1:

The weight of a cycle in a separation graph is non-positive.

Proof: Suppose A is a point in a separation graph such that there is cycle of n edges through the points $B_1, \cdots, B_{n-1}$ back to A. The following inequalities correspond to the edges in the cycle with $I_1, ..., I_n$ as the weights on the edges:

$$A + I_1 \leq B_1$$

$$B_1 + I_2 \leq B_2$$

$$\cdots$$

$$B_{n-1} + I_n \leq A$$

where each I_i is a natural number. We obtain the following inequality by adding the two sides of the above inequalities:

$$A + I \leq A$$

where $I = I_1 + I_2 + \cdots + I_n$. Clearly, I must be non-positive for A to be a point in a computation. Hence, the sum of weights on the transitions forming the cycle is non-positive. □

Definition: The *distance* from point A to point B in a separation graph, *distance* (A, B), is defined to be the longest (largest weight) path from A to B.

The distance from A to B is the relative separation of A and B measured from point A. Intuitively, if A precedes B on a path in a computation tree, then *distance* (A, B) in the separation graph is the earliest relative time B can happen after the occurrence of A; *distance* (B, A) is the earliest relative time A can happen before the occurrence of B (the longest time B can happen after the occurrence of A). As it will be shown in the next theorem, if A precedes B *distance* (A, B) is a non-negative integer and *distance* (B, A) is a non-positive integer.

Refering again to example 1 and Figures 6.3 and 6.4, we will use the above definitions to determine the distance from P_3 to P_5, or the minimum time that must elapse between the completion of action A and the entry of mode M4. The longest path from P_3 to P_5 is the maximum weight of all paths from P_3 to P_5, a non-negative integer. As shown in Figure 6.4, the longest path between the two

points is the path from P_3 to P_1 to P_5. Hence, *distance* (P_3,P_5)=50. The distance from P_2 to P_1 is the largest weight on a path from P_1 to P_2, a non-positive integer. Again, refering to Figure 6.4, the longest path travels from P_2 to P_3 and then to P_1. Hence, *distance* (P_2,P_1)=-80.

Theorem 6.2:

Given two points A and B in a path in a computation tree such that A precedes B,

(a) *distance* $(A,B) \geq 0$, and

(b) *distance* $(B,A) \leq 0$.

Proof:

(a) It is possible that a negative path from A to B exist in a separation graph. However, *distance* (A,B) is defined to be the longest path from point A to point B. Since A precedes B, $A \leq B$, there is a path with the weight of at least zero from A to B in a separation graph. Hence, *distance* $(A,B) \geq 0$.

(b) Suppose the distance from B to A is an integer I. Therefore there is a path from B to A with weight I, so $B + I \leq A$. Furthermore, since A precedes B, we know $A \leq B$ from part (a). Hence, there is a path from B to itself with the weight of I, so

$$B + I \leq B$$

By Theorem 6.1, each cycle in a separation graph must has a non-positive weight. Hence, $I \leq 0$. □

6.4. Generating a Computation Graph

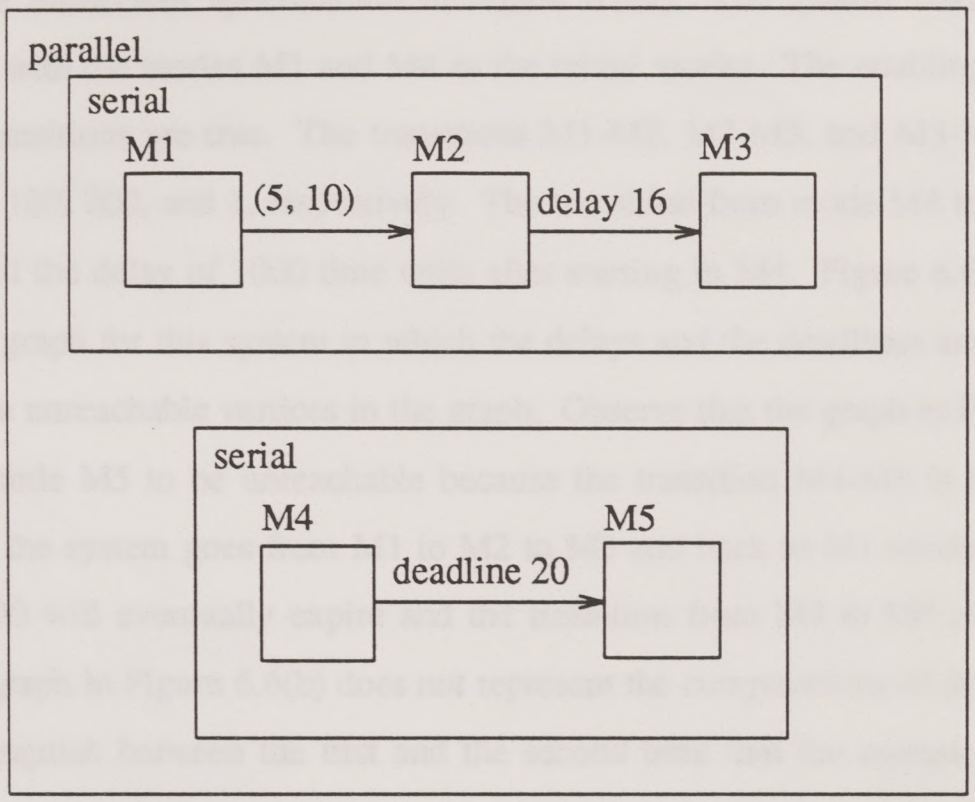
In Section 5, the notion of an augmented computation tree for a Modechart specification was introduced. Since a system computation is often infinite, our goal is to construct a finite *computation graph* that represents the computations

of the system. In the subsequent section, we will describe how to verify a desired property of the system with respect to the corresponding computation graph. Since a mode is control information and may be *viewed* as a state, it is crucial to discuss why a global state reachability graph is not sufficient for our purposes. One reason concerns the timing constraints which are imposed on the actions and transitions. We use two examples to illustrate several key issues in justifying why a global reachability graph is inadequate here.

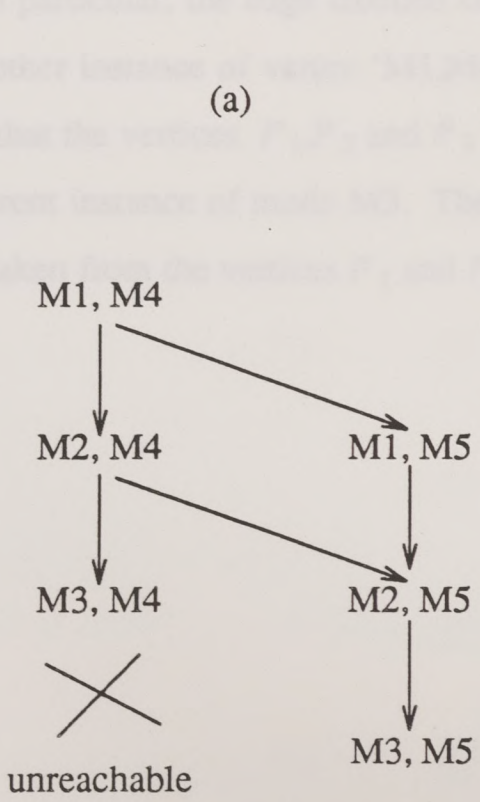
Example 2:

Consider the simple Modechart specification shown in Figure 6.5(a). It consists of two parallel components. The initial modes for the two components are M1 and M4. The enabling conditions for the three transitions in the system are all true. However, the transition from mode M1 to mode M2 must be taken after a delay of 5 and before a deadline of 10 after starting in mode M1. The transition from M2 to M3 must be taken after a delay of 16 time units after entering M2. The transition from M4 to M5 must be taken by a deadline of 20 from starting in mode M4.

Figure 6.5(b) shows a typical global state graph for this system. although the enabling conditions are true, not every global 'state' is reachable. For instance, while the system is in modes M2 and M4, the transition from M2 to M3 cannot be taken before the transition from M4 to M5 because the time after the minimum delay for transition M2-M3 is later than the deadline for the other transition, M4-M5! Hence, it is necessary to consider the relative time ordering to determine what is reachable. In the computation graph proposed at the end of this section, the RTL formulas are used to prune the vertices in the graph which are unreachable due to the timing constraints.



(a)

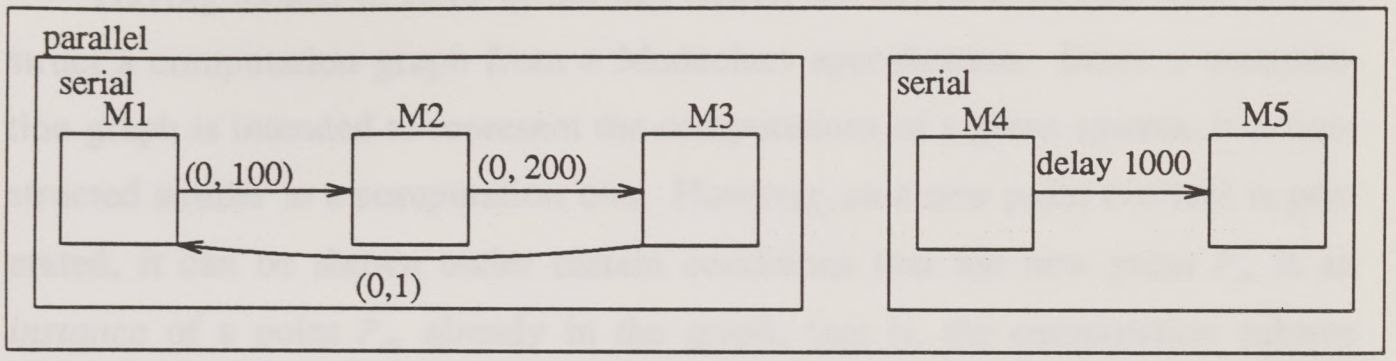


(b)

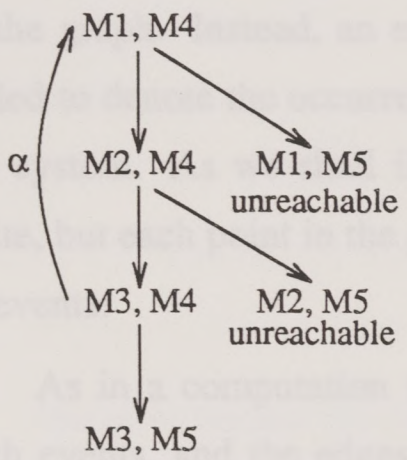
FIGURE 6.5

Example 3:

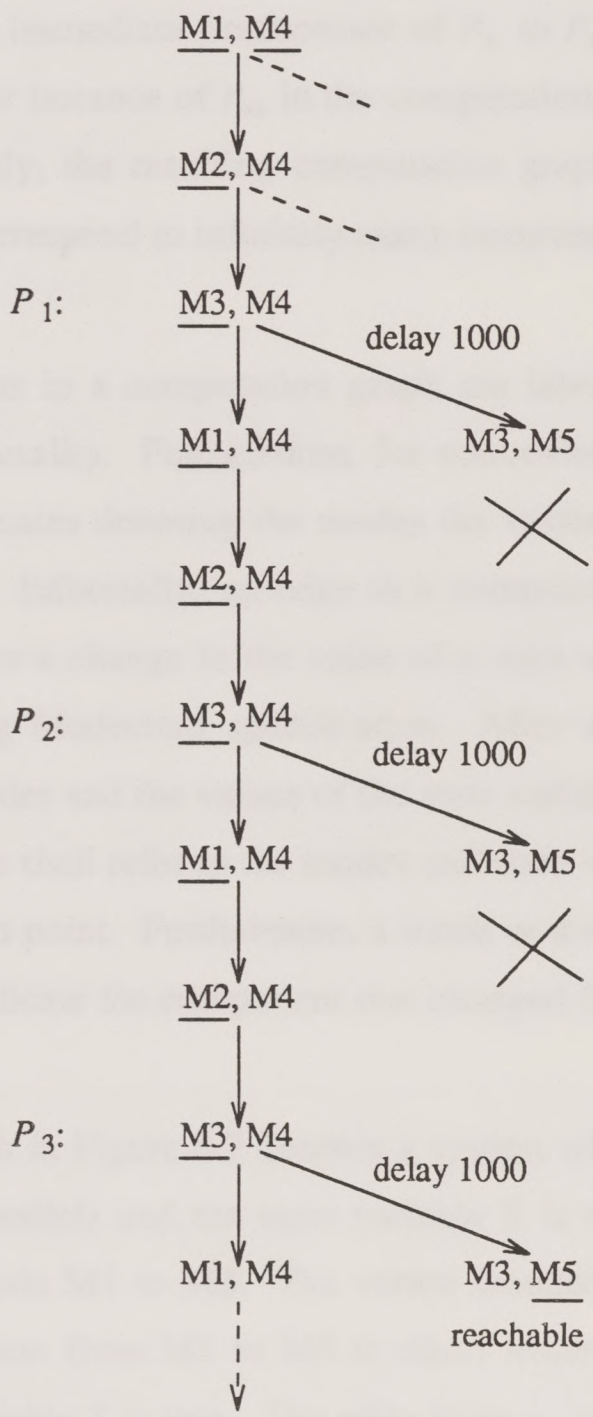
Consider the Modechart specification in Figure 6.6(a). The system consists of two components with the modes M1 and M4 as the initial modes. The enabling conditions on all the transitions are true. The transitions M1-M2, M2-M3, and M3-M1 have the deadlines of 100, 200, and 1, respectively. The transition from mode M4 to M5 cannot be taken until the delay of 1000 time units after starting in M4. Figure 6.6(b) shows a reachability graph for this system in which the delays and the deadlines are considered to discard the unreachable vertices in the graph. Observe that the graph in Figure 6.6(b) shows the mode M5 to be unreachable because the transition M4-M5 is never taken. However, if the system goes from M1 to M2 to M3 and back to M1 several times, the delay of 1000 will eventually expire and the transition from M4 to M5 can be taken. Hence, the graph in Figure 6.6(b) does not represent the computations of the system. It fails to distinguish between the first and the second time that the system entered M1 while it remained in M4. In particular, the edge labelled α in Figure 6.6(b) should go from vertex 'M3,M4' to another instance of vertex 'M1,M4'. Figure 6.6(c) illustrates the revised graph. Observe that the vertices P_1,P_2 and P_3 have the same label. However, each represents a different instance of mode M3. The transition from M4 to M5 which could not have been taken from the vertices P_1 and P_2 can in fact be taken from P_3.


parallel
serial
M1
M2
M3
(0, 100)
(0, 200)
(0,1)
serial
M4
M5
delay 1000

(a)


M1, M4
α
M2, M4
M1, M5
unreachable
M3, M4
M2, M5
unreachable
M3, M5

(b)


P 1:
M1, M4
M2, M4
M3, M4
delay 1000
M1, M4
M3, M5
P 2:
M2, M4
M3, M4
delay 1000
M1, M4
M3, M5
P 3:
M2, M4
M3, M4
delay 1000
M1, M4
M3, M5
reachable

(c)

FIGURE 6.

Having looked at some of the key issues, we can now discuss how to construct a computation graph from a Modechart specification. Since a computation graph is intended to represent the computations of a given system, it is constructed similar to a computation tree. However, as a new point (vertex) is generated, it can be shown under certain conditions that the new point P_n is an *instance* of a point P_m already in the graph; that is, the computation subtree below P_n is identical to the subtree below P_m. Hence, the point P_n is not added to the graph. Instead, an edge from the immediate predecessor of P_n to P_m is added to denote the occurrence of another instance of P_m in the computations of the system. As we shall illustrate shortly, the resulting computation graph is finite, but each point in the graph may correspond to infinitely many occurrences of events.

As in a computation tree, the points in a computation graph are labelled with events, and the edges represent causality. Furthermore, for convenience, each point is labelled with a set of predicates denoting the modes the system is in and the values of the state variables. Informally, an edge in a computation graph corresponds to a mode transition or a change in the value of a state variable as prescribed by the corresponding Modechart specification. After each *point* the system remains in the same modes and the values of the state variables remain the same until the next *point*. We shall refer to the modes and state variables in a point as the *components* of that point. Furthermore, a mode or a state variable is underlined in each point to indicate the component that changed from the previous point.

For example, the computation graph in Figure 6.7 denotes a system which is initially in modes M1 and M3 (in parallel) and the state variable S is true. The first edge takes the system from mode M1 to M2. The vertex labelled p_1 denotes a point at which a mode transition from M1 to M2 is taken while the system is also in mode M3 and state variable S is true. The edge from p_1 to p_2 denotes a change in the value of the state variable S. The vertex labelled p_2

represents a point at which the value of S changed from T to F while the system is in modes M2 and M3.

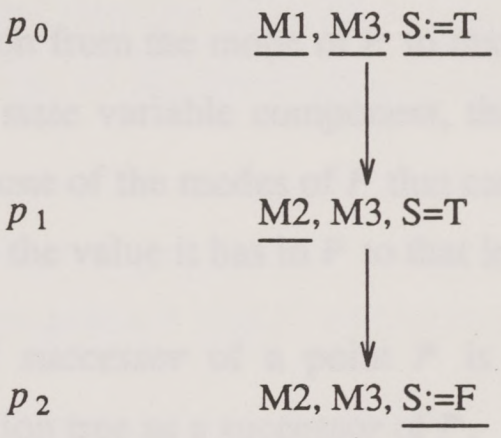


FIGURE 6.7

We now present the conditions under which a point can be shown to be an *instance* of another point in the graph.

Definition: The *event set* of a point P_m is the set of events in P_m or its predecessors such that the events are assigned the same time value in at least one computation of the system:

$$E_m = \{ e \mid e \in P_m \vee [\exists\, P_n \text{ a predecessor of } P_m \wedge e \in P_n \wedge \mathit{distance}(P_n, P_m) = 0] \}$$

Definition: The *event set reference points* for a point P are all of those points Q up to and including P such that $\mathit{distance}(Q, P) = 0$.

Definition: A *reference point for a mode component* M of point P is the point in the computation tree (or graph) closest to P such that $M := T$ is part of the label of that point, and for no points between that point (exclusive) and P (inclusive).

Notice that by this definition, if component M changes at P, then the reference point for M at P is P itself, otherwise, it is the most recent predecessor of P at which M changed.

Definition: The *reference points for a point P* are all of the reference points for the mode components of the point, together with its event set reference points.

Definition: A *potential successor* of a point P is a point that differs from P in at most one component. If that component is a mode component, then there must be a mode transition from the mode of P to that of the potential successor. If that component is a state variable component, then there must be an action that can be executed in one of the modes of P that can result in the state variable changing its value from the value it has in P to that in the potential successor.

Definition: An *actual successor* of a point P is a potential successor that appears in the computation tree as a successor of P.

Note that a potential successor of a point is a point that could be reached if timing requirements were ignored, while actual successors are those that do not violate the real-time constraints.

Definition: Suppose P_m and $P_{m'}$ are two points in a computation graph sharing the same components and event sets. Let P a reference point for P_m and P' be the corresponding reference point for P'_m. Also, suppose A is the set of potential successors of P_m and A′ is the set of potential successors of P'_m, Note that A and A′ are isomorphic since P_m and P'_m have the same mode components and event sets. The *distance equivalence (deq) condition* for reference point P at P_m and P' at P'_m holds if, for each pair of corresponding potential successors $A \in$ A and $A' \in$ A′, either

(a) $distance\,(P,A) = distance\,(P',A')$
$distance\,(A,P) = distance\,(A',P')$

OR the following conditions are true:

(b) If for the reference points P and P' corresponding to a mode component M, $d = \infty$, then

$$distance\,(P,A) \geq r$$
$$distance\,(P',A') \geq r$$

where d is the smallest deadline on any lower/upper bound transition from M, and r is the largest delay on any lower/upper bound transition from M.

AND

(c) If for the reference points P and P' corresponding to a mode component M, $d \neq \infty$, then

$$distance\,(P,A) > d$$
$$distance\,(P',A') > d$$

where d is the smallest deadline on any lower/upper bound transition from M.

AND

(d) If the reference points P and P' correspond to an event from the event set,

$$distance\,(P,A) > 0$$
$$distance\,(P',A') > 0$$

Lemma 6.3 : Given a point P, with potential successors $A_1, A_2, \ldots A_n$, A_i is an actual successor of P if and only if $\forall j$ $distance\,(A_j,A_i) \leq 0$. *Proof:* If the $distance\,(A_j,A_i) > 0$, then the deadline on A_j expires before A_i can occur, so A_i cannot be an actual successor of P, because any computation that took that path would miss the deadline on the transition corresponding to A_j. Likewise, if $distance\,(A_j,A_i) \leq 0$. A_i can occur before the deadline on A_j expires.

Therefore, since all pending deadlines at P are reflected in a potential successor of P, if for a potential successor A_i of P, $\forall$ j *distance* $(A_j,A_i) \leq 0$, then A_i can be a successor of P in some computation, so is an actual successor of P. Conversely, if there is some potential successor A_j of P such that *distance* $(A_j,A_i) > 0$, A_i cannot be an actual successor of P. □

Lemma 6.4:

Suppose P_m and P'_m are two points in a computation graph sharing the same components and the same event sets. Furthermore, suppose the *distance equivalence condition* holds for the corresponding reference points at P_m and P'_m.

(i) A potential successor A of P_m is an actual successor iff the corresponding potential successor A' of P'_m is an actual successor.

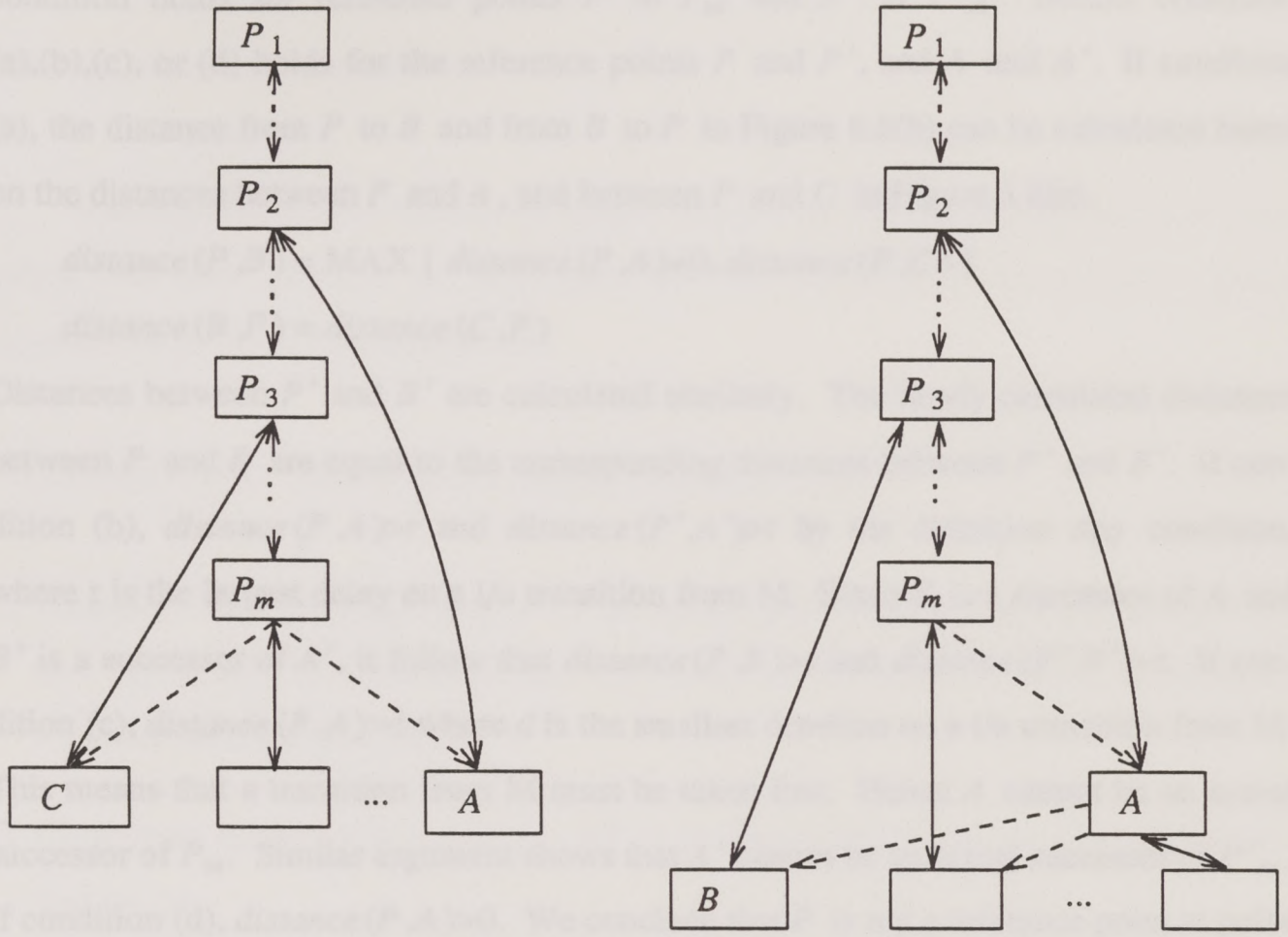
(ii) For each actual successor A of P_m and the corresponding actual successor A' of P'_m, the *deq* condition holds for each reference point at A and A'.

Proof:

(i) Since P_m and P'_m share the same components and the same event sets, for each potential successor A_i of P_m, there is a potential successor A'_i of P'_m such that the two successors share the same components. Since the *deq* condition holds for each mode component at P_m and P'_m, one can show that

$$\textit{distance}\,(A_i,A_j) > 0 \text{ iff } \textit{distance}\,(A'_i,A'_j) > 0$$

where A_i and A_j are potential successors of P_m, and A'_i and A'_j are the corresponding successors of P'_m. It follows from Lemma 6.3 that if a point A is an actual successor of P_m, the corresponding point A' is an actual successor of P'_m.



(a) (b)

FIGURE 6.8

(ii) Let R be the set of reference points at the point P_m. The reference points at A, a successor of P_m, are in the set R$\cup\{A\}$. For each reference point P (both for mode components and event sets) at point A and for each reference point P' at point A', we need to show that one of the four conditions (a through d) in the definition of *deq* condition holds for each B and B', the corresponding potential successors of A and A', respectively. Thus, we need to calculate *distance*(P,B) and *distance*(B,P). (Similar distances between P' and B' must be calculated). There are two cases based on the component that changes at point B. Let Q be the reference point of a component M at point A that changes from A to B.

Case 1: If $Q \in$ R, there is a potential successor of P_m in Figure 6.8(a) such that the same component M changes from P_m to C. From the hypothesis, we know that the *deq*

condition holds for reference points P at P_m and P' at P'_m. Hence, condition (a),(b),(c), or (d) holds for the reference points P and P', and A and A'. If condition (a), the distance from P to B and from B to P in Figure 6.8(b) can be calculated based on the distances between P and A, and between P and C in Figure 6.8(a).

$$distance(P,B) = \text{MAX}\{\ distance(P,A)+0,\ distance(P,C)\ \}$$

$$distance(B,P) = distance(C,P)$$

Distances between P' and B' are calculated similarly. The newly calculated distances between P and B are equal to the corresponding distances between P' and B'. If condition (b), $distance(P,A)>r$ and $distance(P',A')>r$ by the definition *deq* condition, where r is the largest delay on a l/u transition from M. Since B is a successor of A and B' is a successor of A', it follow that $distance(P,B)>r$ and $distance(P',B')>r$. If condition (c), $distance(P,A)>d$ where d is the smallest deadline on a l/u transition from M. This means that a transition from M must be taken first. Hence A cannot be an actual successor of P_m. Similar argument shows that A' cannot be an actual successor of P'_m. If condition (d), $distance(P,A)>0$. We conclude that P is not a reference point at point A, because the event at P is not in the event set of A. Similarly, P' is not reference point at point A'.

Case 2: If $Q=A$, i.e. $Q \notin R$, the reference point of component M at point A is A itself.

$$distance(P,B) = distance(P,A) + W_{A-B}$$

$$distance(B,P) = W_{B-A} + distance(A,P)$$

where W_{A-B} is the weight on the transition from point A to B, and W_{B-A} is the weight on the transition from B to A. The $distance(P',B')$ and $distance(B',P')$ are calculated similarly. For case 2, a similar argument shows that one of the four conditions in the definition of *deq* condition holds for the reference points P and P', and B and B', the successors of A and A', repectively. □

Theorem 6.5:

Given the two points P_m and P'_m in the previous lemma, the subtree generated from point P_m is isomorphic to the subtree generated from point P'_m.

Proof: The theorem can be directly proved from the previous lemma by induction on the level of the subtree generated. □

Having shown the conditions under which two points are instances of each other, we now present an algorithm for constructing a computation graph equivalent to the computation tree.

ALGORITHM: (Computation Graph Construction)

1. Construct P_0, the initial point in the computation graph, denoting the modes that the system is in initially and the values of the state variables. Designate P_0 as an unexpanded point.

2. Choose an unexpanded point P_m, let P_l be the immediate predecessor of P_m if one exists.

3. If there is a potential successor P_r of the point p_l such that *distance* $(P_r, P_m) > 0$, discard P_m.

4. Otherwise, if there is another point P_n in the graph such that P_n and P_m have the same *components* and the same *event sets*, and the *distance equivalence condition* holds for the reference points at P_n and P_m,
   (a) then conclude P_m is another instance of P_n, add an edge from P_l to P_n, and discard P_m.
   (b) Otherwise, let $\tau_1, \tau_2, \cdots, \tau_k$ be the set of transitions that can be taken from P_m with potential successors $Q_1, Q_2, \cdots, Q_k$. Add each Q_i to the graph, mark it as an unexpanded point and add an edge from the point P_m to Q_i. Mark P_m as an expanded point.

5. Repeat steps 2 through 4 until there are no unexpanded points left.

Lemma 6.6: The computation graph is finite.

Proof : Partition the points of the computation graph by their mode components and event sets. For each mode, refine the partition as follows. If the smallest deadline on any transition from the mode is ∞, then the refinement partitions the points into classes 0 through r, where r is the largest delay on a transition from the mode, according to the value of *distance* (P',P), where P' is the reference point for P for the given mode component, with any value over r in the class of r, because they all satisfy the deq condition for the component of P'. If the smallest deadline on any transition from P' is finite, say d, then for each potential successor A of P,choose the classes to be 0 through $d+1$, and allocate the points by the value of *distance* (P',A). Further refine the partition by the values of *distance* (A,P') for each A and P'. Alternatively, if the deadline on A is finite, then the *distance* (A,P') is bounded by the sum of the two deadlines. Therefore, this refinement also introduces only a finite number of equivalence classes.

The above partition divides the point space into a finite number of equivalence classes, and within each class, the points pairwise satisfy the requirements of Theorem 5, so the computation graph will contain at most one copy of a point from each equivalence class. Therefore the graph is finite. □

6.5. Example: Railroad Crossing

In this section, a Modechart specification for a system which monitors and controls a railroad crossing is presented. A safety assertion for the system is proved using the approach presented here. This example is based on one that appears in [Leveson & Stolzy 85].

6.5.1. Specification

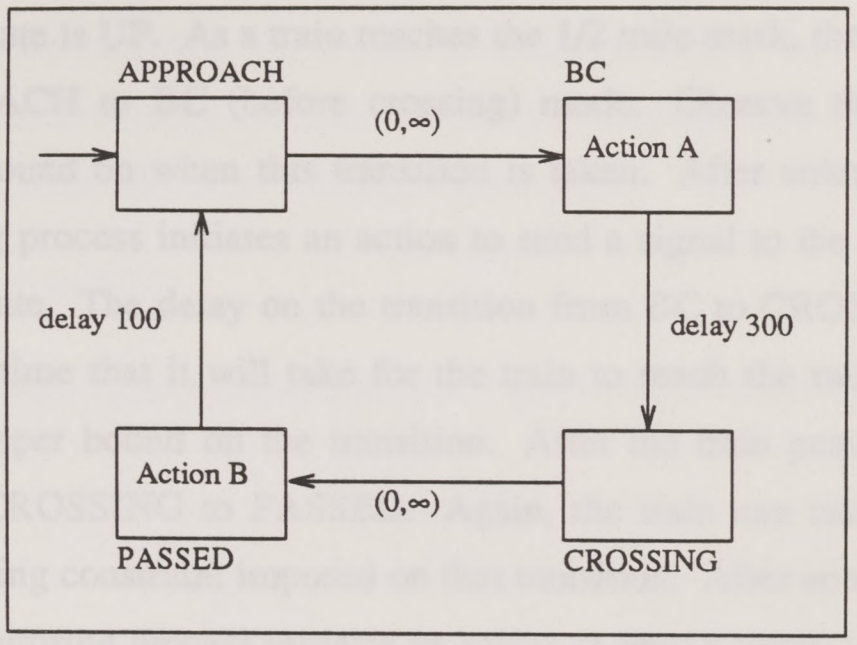
Figure 6.9 illustrates a Modechart specification for a railroad crossing. The system consists of two parallel components: a process monitoring the oncoming trains and a controller interface to the guard gate. The monitoring process is in one of four modes, depending on its view of the train:

APPROACH:	The train is approaching, but very far from, the crossing.
BC:	The train is less than a given distance, 1/2 mile, from the crossing. The monitor sends a signal to controller process to lower the gate.
CROSSING:	The train is within the railroad crossing.
PASSED:	The train has cleared the crossing, and the monitor sends a signal to the controller process to raise the gate.

The controller process for the gate can be in one of the following modes:

UP:	The gate is up.
MOVEDOWN:	Signal is received to lower the gate, and an action is initiated to lower the gate.
DOWN:	The gate is down.
MOVEUP:	Signal is received to raise the gate, and an action is initiated to raise the gate.

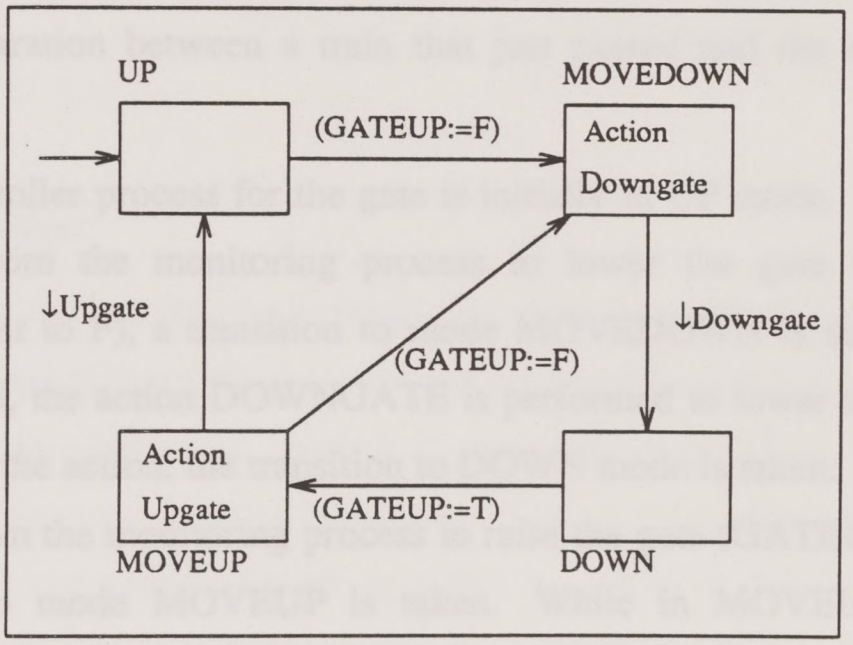
MONITOR



Action A: sets (GATEUP:=F), deadline = 50

Action B: sets (GATEUP:=T), deadline = 50

GATE-CONTROLLER



Action DOWNGATE:	set light to red move gate down deadline = 50
Action UPGATE:	move gate up; set light to green deadline = ∞

Initially, (GATEUP:=T)

FIGURE 6.9

Initially, the monitoring process is in APPROACH mode waiting for a train and the gate is UP. As a train reaches the 1/2 mile mark, the system moves from APPROACH to BC (before crossing) mode. Observe that there is no lower/upper bound on when this transition is taken. After entering BC mode, the monitoring process initiates an action to send a signal to the gate controller to lower the gate. The delay on the transition from BC to CROSSING denotes the minimum time that it will take for the train to reach the railroad crossing. There is no upper bound on the transition. After the train passes, the system moves from CROSSING to PASSED. Again, the train can take its time and there is no timing constraint imposed on that transition. After entering PASSED mode, the monitoring process initiates an action to send a signal to the gate controller to raise the gate. Finally, the system can move from PASSED to APPROACH after the delay of 100 to reflect that another train may be approaching. The timing constraint on this transition indicates that there is a minimum separation between a train that just passed and the next oncoming train.

The controller process for the gate is initially in UP mode. When a signal is received from the monitoring process to lower the gate (state variable GATEUP is set to F), a transition to mode MOVEDOWN is taken. While in MOVEDOWN, the action DOWNGATE is performed to lower the gate. Upon completion of the action, the transition to DOWN mode is taken. When a signal is received from the monitoring process to raise the gate (GATEUP is set to T), a transition to mode MOVEUP is taken. While in MOVEUP, the action UPGATE is done to move the gate up. Upon completion of the action, the transition to UP mode is taken. While executing the action Upgate in mode MOVEUP, if another signal is received to lower the gate, the action is aborted and a transition to mode MOVEDOWN is taken.

6.5.2. Safety assertion

In the above example, the system is unsafe if a train can cross while the gate is up. Hence, a safety assertion for the above system requires that the gate must be down while a train is crossing, Figure 6.10.

Safety Assertion in RTL:

$$\forall i \exists j \; @((\text{DOWN}:=T),j) \leq @((\text{CROSSING}:=T),i) \wedge @((\text{CROSSING}:=F),i) \leq @((\text{DOWN}:=F),j)$$

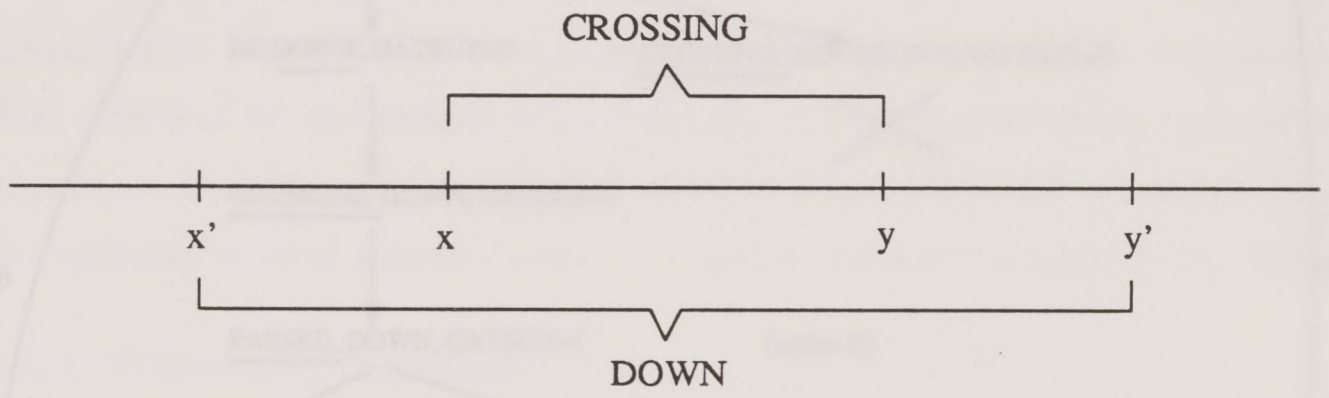


FIGURE 6.10

6.5.3. Verification

Figure 6.11 illustrates the computation graph for the railroad crossing system. The unreachable points (vertices) are shown to be unreachable by applying Lemma 6.3. One edge in the graph is of particular interest because it illustrates how deq condition and Lemma 5 are used. Consider the edge labelled α in the Figure 6.11. It is a backward edge to vertex P_i. Observe that it was not necessary to create the point P'_i because P'_i is an instance of P_i by Lemma 5. The mode components of P_i and P'_i are the same, and the event set for each point is {GATEUP:=F}. Furthermore, the corresponding reference points at P_i and P'_i satisfy the deq condition. Therefore, Lemma 5 guarantees that the subtree below P_i is isomorphic to the subtree below P'_i. A similar argument produces the edge labelled β.

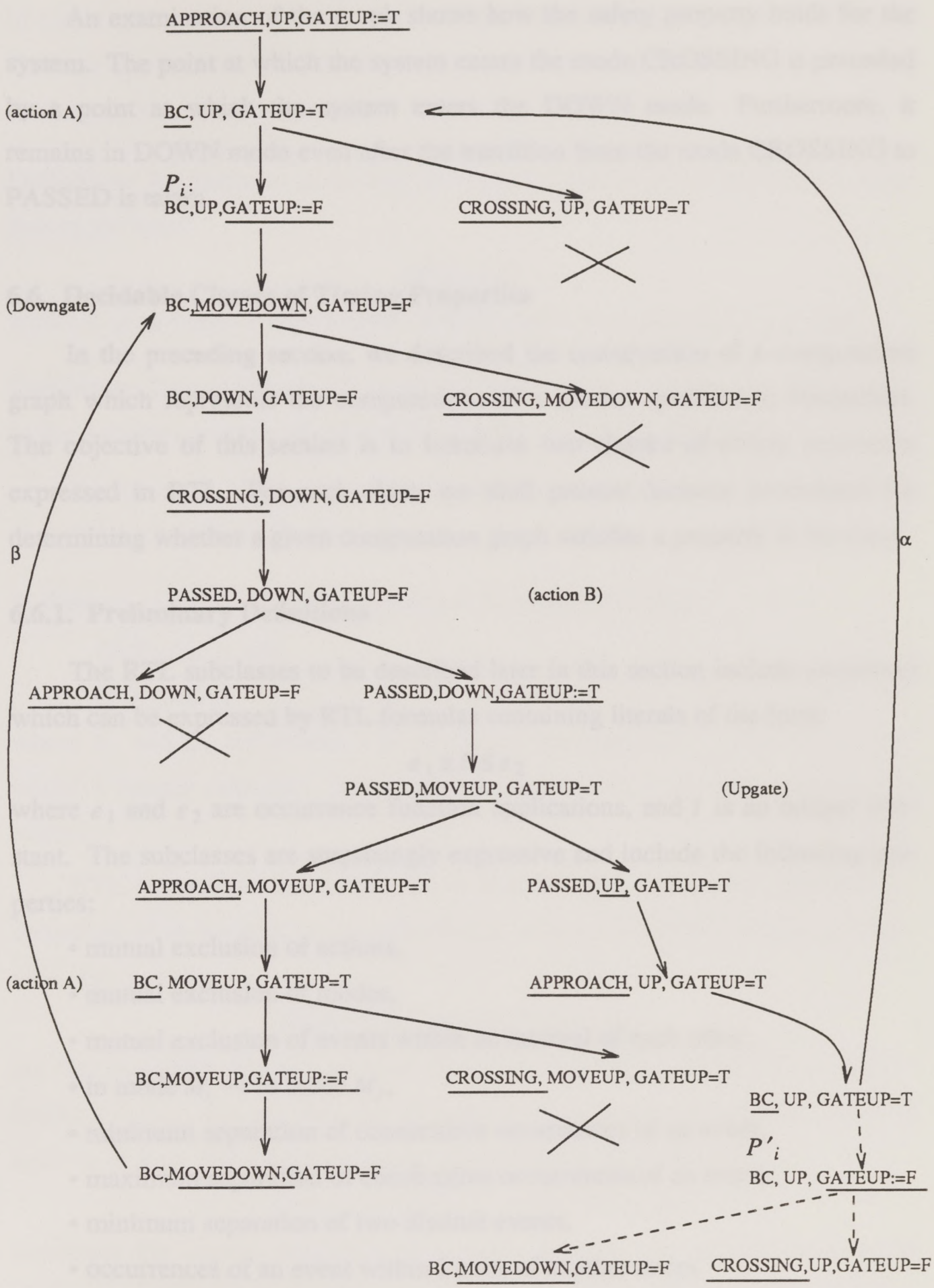

APPROACH,UP,GATEUP:=T
(action A)
BC, UP, GATEUP=T
P_i:
BC,UP,GATEUP:=F
CROSSING, UP, GATEUP=T
(Downgate)
BC,MOVEDOWN, GATEUP=F
BC,DOWN, GATEUP=F
CROSSING, MOVEDOWN, GATEUP=F
CROSSING, DOWN, GATEUP=F
β
α
PASSED, DOWN, GATEUP=F
(action B)
APPROACH, DOWN, GATEUP=F
PASSED,DOWN,GATEUP:=T
PASSED,MOVEUP, GATEUP=T
(Upgate)
APPROACH, MOVEUP, GATEUP=T
PASSED,UP, GATEUP=T
(action A)
BC, MOVEUP, GATEUP=T
APPROACH, UP, GATEUP=T
BC,MOVEUP,GATEUP:=F
CROSSING, MOVEUP, GATEUP=T
BC, UP, GATEUP=T
P'_i
BC,MOVEDOWN,GATEUP=F
BC, UP, GATEUP:=F
BC,MOVEDOWN,GATEUP=F
CROSSING,UP,GATEUP=F

FIGURE 11.

An examination of the graph shows how the safety property holds for the system. The point at which the system enters the mode CROSSING is preceded by a point at which the system enters the DOWN mode. Furthermore, it remains in DOWN mode even after the transition from the mode CROSSING to PASSED is taken.

6.6. Decidable Classes of Timing Properties

In the preceding section, we described the construction of a computation graph which represents the computations of a system specified in Modechart. The objective of this section is to introduce two classes of timing properties expressed in RTL. For each class, we shall present decision procedures for determining whether a given computation graph satisfies a property in the class.

6.6.1. Preliminary Definitions

The RTL subclasses to be described later in this section include properties which can be expressed by RTL formulas containing literals of the form:

$$e_1 \pm I \leq e_2$$

where e_1 and e_2 are occurrence function applications, and I is an integer constant. The subclasses are surprisingly expressive and include the following properties:

- mutual exclusion of actions,
- mutual exclusion of modes,
- mutual exclusion of events within an interval of each other,
- in mode $M_i \rightarrow$ in mode M_j,
- minimum separation of consecutive occurrences of an event,
- maximum separation of consecutive occurrences of an event,
- minimum separation of two distinct events,
- occurrences of an event within I units of another event.

The two subclasses and the corresponding verification procedures are described

in terms of intervals and endpoints. Informally, an endpoint is an occurrence function application, i.e., the time of occurrence of an event. An interval is denoted by two related endpoints.

Definition: An *endpoint* is defined to be an application of an occurrence function of the form

$$@(E,i\pm c)$$

where E is an event, i is an integer variable, and c is a non-negative integer.

Definition: Two endpoints are *related* if both contain the same integer variable in the occurrence index of their respective '@' function applications.

For example, $@(E_1,j)$ and $@(E_2,j+2)$ are related endpoints, but $@(\downarrow A,i)$ and $@(\downarrow A,j)$ are not related.

Definition: An *interval* consists of two endpoints of the form

$@(E_1,i\pm c_2)$, and

$@(E_2,i\pm c_2)$

where E_1 and E_2 are events, i is an integer variable, and c_1 and c_2 are non-negative integer constants. Two endpoints denoting an interval are by definition related.

RTL inequalities are employed to describe the relationship between the endpoints of an interval or the relationship between endpoints of distinct intervals. For instance,

$$@((M:=T),i) \text{ and } @((M:=T),i+1)$$

are two endpoints defining an interval, and the inequality $@((M:=T),i) + 100 \leq @((M:=T),i+1)$ denotes that the successive occurrences of entering mode M must be at least 100 time units apart. Similarly, the RTL formula

$$\forall i \forall j \; @(\downarrow A,i) \leq @(\uparrow B,j) \vee @(\downarrow B,j) \leq @(\uparrow A,i)$$

expresses the mutual exclusion relationship between the interval denoted by $@(\uparrow A,i)$ and $@(\downarrow A,i)$ and the interval denoted by $@(\uparrow B,j)$ and $@(\downarrow B,j)$, i.e., mutual exclusion of actions A and B.

The two classes of timing properties to be described shortly are defined in terms of endpoints and intervals. Suppose we are given a computation graph and two endpoints @(E_1,i) and @(E_2,i) whose relationship is to be verified with respect to the graph. If each cycle in the graph is labeled with the same number of E_1 and E_2 events, then by designating a point in the graph containing E_1 as representing the ith occurrence of E_1, the point representing the ith occurrence of E_2 follows in a bounded number of iterations of cycles of the graph. The number of iterations must reflect both the number of additional occurrences of E_1 before the cycle is entered, as well as the difference in the constants. However, if a cycle is labeled with different number of E_1 and E_2 events, then the ith occurrence of E_1 is potentially a number of cycles from the ith occurrence of E_2 that is bounded by a function of i.

Definition: Suppose G is a computation graph, @(E_1,$i \pm c_1$) and @(E_2,$i \pm c_2$) are related endpoints. A cycle in G is said to *preserve* the endpoints if the number of vertices in the cycle labeled with E_1 is the same as the number of vertices labeled with E_2. The graph is said to *preserve* an RTL formula if each cycle in the graph preserves each set of related endpoints in the formula.

In the remainder of this paper, we are interested in the RTL formulas which are preserved by a given computation graph. This does not appear to be a very restrictive condition. It is trivially true that when E_1 and E_2 are the same in the above definition, a computation graph *preserves* the endpoints. Furthermore, when E_1 and E_2 are the entry and exit events for the same mode, the related endpoints containing E_1 and E_2 are preserved by a computation graph of a Modechart specification. The same holds when E_1 and E_2 denote transition events for a state variable, or when they are the start and stop events for an action. Hence, for the above cases, it is trivial to check if each set of related points in an RTL formula are preserved by a computation graph. However, when two related points do not fit in the above cases, the check for the condition can be performed in polynomial time. The remaining subsections introduce two

subclasses of timing properties expressed in RTL for which one can decide if a computation graph satisfies the property.

6.6.2. Class 1: Minimum/Maximum Separation of Related Endpoints

In this subsection we are concerned with a class of timing properties which specify the relative and absolute ordering of related endpoints. The class consists of the RTL formulas of the form Q F where Q is either $\forall i$ or $\exists i$, and F is a quantifier free formula such that each inequality in F is of the form

$$e_1 \pm I \leq e_2$$

where e_1 and e_2 are related endpoints, and I is a natural number. Observe that the endpoints in F are related because the same variable (plus/minus an integer offset) appears as the occurrence index in each endpoint. For example, the following formula states that two successive entries to the mode M are either 100 time units apart or within 50 time units of each other:

$$\forall i \;\; @((M:=T),i) + 100 \leq @((M:=T),i+1) \;\vee\; @((M:=T),i+1) \leq @((M:=T),i) + 50$$

As a different example, consider a formula specifying that an instance of action A is performed once each time the system is in mode M:

$$\forall i \;\; @((M:=T),i) \leq @(\uparrow A,i) \;\wedge\; @(\downarrow A,i) \leq @((M:=F),i)$$

Each inequality in a formula of class 1 can be rewritten as one of two forms:

$$e_1 - e_2 \leq I \qquad (1)$$

$$I \leq e_2 - e_1 \qquad (2)$$

The first inequality specifies a maximum separation of I on the two endpoints, and the second inequality imposes a minimum separation of I on the endpoints.

Suppose G is a computation graph and F is a *class 1* RTL formula which is preserved by G. The objective is to determine if every computation in the graph G satisfies the timing property expressed in the formula F. Since F has one

quantifier, all the endpoints in the formula form one set S of related endpoints which are preserved by every cycle in G:

$$S = \{ @(E_1, i+c_1), @(E_2, i+c_2), \ldots, @(E_n, i+c_n) \}$$

where i is the occurrence index variable quantified by a $\forall$ or $\exists$, and n is the number of endpoints in F. As mentioned in subsection 8.1. if we take a point P in G labeled with an event E_j (for some $1 \leq j \leq n$) to mean the ith occurrence of E_j, then all the corresponding end points in F (if any exist) can be located by unrolling each cycle in G a constant number of times. For each inequality in F, if it is of form (1), we find the maximum distance between the two points in the unrolled graph and assign a truth value to the inequality. Otherwise, the inequality is of form (2). Hence, we find the minimum distance between the two points in the unrolled graph and assign a truth value to the inequality. In this manner, a truth value can be obtained for F. However, this truth value applies to all of the occurrences (perhaps infinitely many) corresponding to the point P in the graph.

If the formula F is quantified by a $\forall$, then the above procedure must be repeated for each point in the graph G which is labeled with the event E_j. A *true* assignment to the formula F in each case is necessary to conclude that every computation in G satisfies F. However, if F is quantified by a $\exists$, then it is necessary to show that F is assigned *true* for at least one application of the procedure to a point labeled with the event E_j.

6.6.3. Class 2: Exclusion/Inclusion of Interval and Endpoints

In this subsection we are concerned with a class of timing properties describing exclusion/inclusion of an endpoint or an interval with respect to another interval. The timing properties in this class can be expressed as RTL inequalities relating the endpoints of an interval to the endpoints of another interval. Hence, the RTL formulas specifying the properties in this class have two quantifiers, one for each interval. Recall that an endpoint was defined in

subsection 8.1. to be of the form $@(E,i\pm c)$. We extend that definition to include an integer offset I:

$$@(E,i\pm c) \pm I$$

The properties in this class can be in two general forms. Suppose e_1 and e_2 are the endpoints of an interval α and e_3 and e_4 are the endpoints of a second interval β. The first form specifies the inclusion of one interval inside the other:

$$Q \;\; e_1 \le e_3 \wedge e_4 \le e_2 \qquad (1)$$

where Q is a prefix of two quantifiers on the occurrence index variables for the intervals. For instance, the following RTL formula states that each stay in mode M contains some execution of action A which starts at least 100 time units after entering the mode:

$$\forall i \exists\!\!\!| j \;\; @((M:=T),i) + 100 \le @(\uparrow A,j) \wedge @(\downarrow A,j) \le @((M:=F),i)$$

The second form specifies the exclusion of one interval from the other:

$$Q \;\; e_4 \le e_1 \vee e_2 \le e_3 \qquad (2)$$

For instance, the following formula specifies the mutual exclusion of two actions A and B:

$$\forall i \forall j \;\; @(\downarrow B,j) \le @(\uparrow A,i) \vee @(\downarrow A,i) \le @(\uparrow B,j)$$

Given a computation graph G and a RTL formula F specifying a timing property in class 2, the objective is to determine if every computation in G satisfies F. We identify five cases for the inclusion property shown in form (1). In each of the cases below, i is the occurrence index variable for the endpoints e_1 and e_2 (interval α), and j is the variable for the endpoints e_3 and e_4 (interval β).

(1a) $\forall i \exists\!\!\!| j \;\; e_1 \le e_3 \wedge e_4 \le e_2$

This case specifies that *every* instance of interval α contains *some* instance

of interval β. The above formula is verified by checking each appearance of interval α in G for the existence of interval β.

(1b) $\forall i \exists j \; e_3 \leq e_1 \wedge e_2 \leq e_4$

This case specifies that *every* instance of interval α is contained in *some* instance of interval β. The above formula is verified by first locating an appearance of interval α in G, then checking if an instance interval β surrounds it. Observe that only a finite number of instances of β needs to be checked because there can be only a finite number of pairs of points in the graph labeled with the events in e_3 and e_4 such that it encloses the points corresponding to each appearance of α. The above procedure must be repeated for each appearance of interval α in graph G.

(1c) $\exists i \exists j \; e_1 \leq e_3 \wedge e_4 \leq e_2$

This case specifies that some instance of interval α contains some instance of interval β.

(1d) $\exists i \forall j \; e_1 \leq e_3 \wedge e_4 \leq e_2$

This case specifies that some instance of interval α contains every instance of interval β.

(1e) $\exists i \forall j \; e_3 \leq e_1 \wedge e_2 \leq e_4$

This case specifies that *some* instance of interval α is contained in *every* instance of interval β.

Similarly, we identify four cases for the exclusion property shown in form (2). In each of the cases below, i is the occurrence index variable for the endpoints e_1 and e_2 (interval α), and j is the variable for the endpoints e_3 and e_4 (interval β).

(2a) $\forall i \forall j \; e_4 \leq e_1 \vee e_2 \leq e_3$

This case specifies that *every* instance of interval α is mutually exclusive with *every* instance of β. Cases 2b-2d are defined similarly below.

(2b) $\exists i \exists j \; e_4 \leq e_1 \vee e_2 \leq e_3$

(2c) $\forall i \exists j \; e_4 \leq e_1 \vee e_2 \leq e_3$

(2d) $\exists i \forall j \; e_4 \leq e_1 \vee e_2 \leq e_3$

The two classes of timing properties above obviously do not include all the properties which one may wish to specify about a Modechart system. For example, the following properties cannot be expressed in either of the above classes:

- Each instance of an interval α contains an instance of interval β or an instance of interval γ.
- The three intervals α, β, and γ each contain the next in the list.

For both properties, alternative decision procedures can be developed to verify the properties with respect to a graph. An area of continuing research is the discovery of other decidable classes of timing properties.

6.7. Concluding Remarks

The verification technique proposed in this chapter requires the construction of a transition graph for a given Modechart specification. System properties are proved using the transition graph and the corresponding RTL formulas. Although timing constraints are used to discard unreachable vertices, constructing the entire transition graph suffers from the same drawbacks as similar techniques based on reachability analysis. In particular, it is impractical to generate the entire graph for a complex Modechart specification. A natural extension of the proposed approach is to investigate the verification of properties by constructing a partial graph.

Chapter 7

Concluding Remarks and Future Research

The focus of this work has been on the modeling and verification of timing properties in Real-Time Systems. The approach is based on two languages: Real Time Logic, which is especially suitable for the specification of the relative and absolute timing of events, and Modechart, a high-level specification language for real-time systems. When a system specification and a timing property are expressed as RTL formulas, the verification is performed by showing that the property is a theorem derivable from the specification. Alternatively, a Modechart specification is shown to satisfy a timing property expressed in RTL by constructing a graph which denotes the computations of the system; then the property is checked to be satisfied by every computation in the transition graph.

On the subject of the verification of timing properties of Modechart Specifications, two immediate extensions of this work come to mind. One way of extending this work is to explore other classes of timing properties for which decidability can be obtained. Another way to extend this work is by investigating the verification of properties by generating partial transition graphs. Although timing constraints are used to discard unreachable vertices, building the entire transition graph suffers from the same drawbacks as similar techniques based on reachability analysis. This makes it impractical to generate the entire graph for a complex Modechart specification.

Other avenues of future research relate to the logic introduced in this work. Since fault-tolerance is an important requirement in many time-critical systems, it is relevant to examine the suitability of Real Time Logic for modeling fault-tolerant systems. An alternative direction for future research would be to consider the applicability of the approach to the modeling and verification of hardware.

Bibliography

[Alford 77].

M.W. Alford, ''A Requirements Engineering Methodology for Real-Time Processing Requirements,'' *IEEE Trans. on Software Engineering* **SE-3**(1)(Jan. 1977).

[Bernstein & Harter 81].

A. Bernstein and P.K. Harter, ''Proving Real-Time Properties of Programs with Temporal Logic,'' *Proc. 8th Symposium on Operating System Principles, ACM SIGOPS*, pp. 1-11 (1981).

[Bledsoe 74].

W.W. Bledsoe, ''The SUP-INF Method in Presburger Arithmetic,'' ATP-18, Dept. of Math., University of Texas at Austin, Austin, Texas (Dec. 1974).

[Bledsoe & Hines 80].

W.W. Bledsoe and L.M. Hines, ''Variable Elimination and Chaining in a Resolution-based Prover for Inequalities,'' pp. 70-87 in *5th Conf. on Automated Deduction: Lecture Notes in Computer Science*, ed. R. Kowalski,Springer-Verlag, New York, N.Y. (1980).

[Bledsoe el al 81].

W.W. Bledsoe, R. Neveln, and R. Shostak, ''Some Completeness Results for a Class of Inequality Provers,'' ATP-60, University of Texas, Austin, Texas (March 1981).

[Bundy 83].

A. Bundy, *The Computer Modelling of Mathematical Reasoning*, Academic Press, London (1983).

[Chang & Lee 73].

C. Chang and R.C. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, N.Y. (1973).

[Cooper 71].

D.C. Cooper, ''Programs for Mechanical Program Verification,'' in *Machine Intelligence 6*, ed. D. Michie,Edinburgh Univ. Press, Edinburgh (1971).

[Cooper 72].

D.C. Cooper, ''Theorem Proving in Arithmetic without Multiplication,'' in *Machine Intelligence 7*, ed. D. Michie,Edinburgh Univ. Press, Edinburgh (1972).

[Dasarathy 85].

B. Dasarathy, ''Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them,'' *IEEE Transactions on Software Engineering* **SE-11**(1)(Jan. 1985).

[Davis & Vick 77].

C.G. Davis and C.R. Vick, ''The Software Development System,'' *IEEE Trans. on Software Engineering* **SE-3**(1) pp. 69-84 (Jan. 1977).

[Downey 72].

Peter J. Downey, ''Undecidability of Presburger Arith. with a Single Monadic Predicate Letter,'' Center for Research in Computing Technology 18-72, Harvard University (1972).

[Harel 86].

D. Harel, ''Statecharts: A Visual Formalism for Complex Systems,'' The Weizmann Institute of Sci. Tech. Report, Israel (July 1986). (also in Science of Programming 8, 1987)

[Harel et al 87].

D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman, ''On the Formal Semantics of Statecharts,'' *Proc. 2nd Symp. on Logic in Computer Science*, pp. 54-64 (June 1987).

[Heninger 80].

K.L. Heninger, ''Specifying Software Requirements for Complex Systems: New Techniques and their Application,'' *IEEE Transactions on Software Engineering*, **SE-6**(1) pp. 2-13 (Jan. 1980).

[Jahanian & Mok 86b].

F. Jahanian and A.K. Mok, ''A Graph-Theoretic Approach for Timing Analysis in Real Time Logic,'' *Proc. of Real-Time Systems Symposium*, pp. 98-108 (Dec. 1986).

[Jahanian & Mok 86a].

F. Jahanian and A.K. Mok, ''Safety Analysis of Timing Properties in Real-Time Systems,'' *IEEE Transactions on Software Engineering* **SE-12**(9) pp. 890-904 (Sept. 1986).

[Jahanian & Mok 87].

F. Jahanian and A.K. Mok, ''A Graph-Theoretic Approach for Timing Analysis and its Implementation,'' *IEEE Transactions on Computers* **C-36**(8) pp. 961-975 (August 1987).

[Jahanian et al 88a].

F. Jahanian, A. Mok, and D. Stuart, ''Formal Specification of Real-Time Systems,'' Department of Computer Sciences, TR-88-25, University of Texas at Austin (June 1988).

[Jahanian & Mok 88].

F. Jahanian and A.K. Mok, ''Modechart: A Specification Language for Real-Time Systems,'' *to appear in IEEE Transactions on Software Engineering*, (1988).

[Jahanian et al 88b].

F. Jahanian, R.S. Lee, and A.K. Mok, ''Semantics of Modechart in Real Time Logic,'' *Proc. 21st Hawaii International Conference on System Sciences*, (Jan. 1988.).

[Jain & Lam].

P. Jain and S.S. Lam, ''Modeling and Verification of Real-Time Protocols for Broadcast Networks,'' *IEEE Trans. on Software Engineering*, pp. 924-37 (August, 1987).

[Lamport 83].

L. Lamport, ''What Good is Temporal Logic?,'' pp. 657-668 in *Proceeding of IFIP, Information Processing*, ed. R.E.A. Mason,,North-Holland (1983).

[Lawler 76].

E.L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, N.Y. (1976).

[Leveson & Stolzy 85].

N. G. Leveson and J. L. Stolzy, ''Analyzing Safety and Fault Tolerance Using Time Petri Nets,'' *TAPSOFT: Joint Conference on Theory and Practice of Software Development*, Springer Verlag, (March 1985).

[Luqi & Berzins 87].

Luqi and V. Berzins, ''Rapid Prototyping of Real-Time Systems,'' Computer Science Dept. Tech. Report 87-5, Naval Postgraduate School, Monterey, Calif. (1987).

[Lusk et al 82a].

E.L. Lusk, W. McCune, and R.A. Overbeek, ''Logic Machine Architecture: Inference Mechanism,'' pp. 85-108 in *Proc. of the Sixth Conf. on Automated Deduction: Lecture Notes in Computer Science, Vol. 138*, ed. E.W. Loveland,Springer-Verlog, New York, N.Y. (1982).

[Lusk et al 82b].

E.L. Lusk, W. McCune, and R.A. Overbeek, ''Logic Machine Architecture: Kernel Functions,'' pp. 70-84 in *Proc. of the Sixth Conf. on Automated Deduction: Lecture Notes in Computer Science, Vol. 138*, ed. E.W. Loveland,Springer-Verlog, New York, N.Y. (1982).

[Merlin & Farber 76].

P.M. Merlin and D.J. Farbar, ''Recoverability of Communication Protocols-Implications of a Theoretical Study,'' *IEEE Trans. on Communications*, pp. 1036-43 (Sept. 1976).

[Mok 1983].

A.K. Mok, ''Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment,'' Ph.D. Dissertaion, MIT, Boston, Mass (May, 1983).

[Mok 85].

A.K. Mok, ''SARTOR-a Design Environment for Real-Time Systems,'' *Proc. 9th IEEE COMPSAC*, pp. 174-181 (Oct. 1985).

[Ostroff 87].

J. S. Ostroff, ''Real-time Computer Control of Discrete Event Systems Modelled by Extended State Machines: a Temporal Logic Approach,'' Ph.D. Dissertation, Department of Electrical Engineering, University of Toronto,

Toronto, Canada (January, 1987).

[Ostroff & Wonham 87].

J. S. Ostroff and W. M. Wonham, ''Modelling, Specifying and Verifying Real-time Embedded Computer Systems,'' *Proceedings of the Real Time Systems Symposium*, pp. 124-32 (December, 1987).

[Parnas et al 78].

D. Parnas, K.L. Heninger, J.W. Kallander, and J.E. Shore, ''Software Requirements for the A-7E Aircraft,'' NRL memorandum report 3876, Washington, D.C. (Nov. 1978).

[Ramchandani 74].

C. Ramchandani, ''Analysis of Asynchronous Concurrent Systems by Timed Petri Nets,'' TR 120, Project MAC, MIT (Feb. 1974).

[Razouk & Phelps 84].

R.R. Razouk and C.V. Phelps, ''Performance Analysis of Timed Petri Nets,'' *Proc. of 4th International Workshop on Protocol Spec, Verf and Testing*, (June 1984).

[Sahni 85].

S. Sahni, *Concepts in Discrete Mathematics, Second Edition,* Camelot, Fridley, Minnesota (1985).

[Schwartz et al 83].

R.L. Schwartz, M. Melliar-Smith, and F.H. Vogt, ''An Interval Logic for Higher-Level Temporal Reasoning,'' *Proc. Second Annual ACM Symposium on Principles of Distributed Computing*, pp. 173-185 (1983).

[Shankar & Lam 86].

A.U. Shankar and S.S. Lam, ''Time-Dependent Distributed Systems: Proving Safety, Liveness and Real-Time Properties,'' Tech. Report TR-85-24 (revised),, Computer Science Department, Univ. of Texas, Austin, Texas (Oct. 1986).

[Shostak 77].

R.E. Shostak, ''On the SUP-INF Method for Proving Presburger Formulas,'' *JACM*, **24**(4) pp. 529-543 (Oct. 1977).

[Shostak 79].

R.E. Shostak, ''A Practical Decision Procedure for Arithmetic with Function Symbols,'' *JACM*, **26**(2) pp. 350-361 (April 1979).

[Wos et al 84].

L. Wos, R. Overbeek, E. Lusk, and J. Boyle, *Automated Reasoning*, Prentice-Hall, Englewood Cliffs, N.J. (1984).

[Zave 85].

P. Zave, ''A Distributed Alternative to Finite-State-Machine Specifications,'' *ACM Trans. Prog. Lang. Syst.* **7** pp. 10-36 (1985).

[Zuberek 1980].

W.M. Zuberek, ''Timed Petri Net and Preliminary Performance Eveluation,'' *7th Annual Symposium on Computer Architecture*, pp. 88-96 (1980).

The vita has been removed from the digitized version of this document.